A Byte of Python

Swaroop H

目錄

简明 Python 教程

版权信息

Swaroop, C. H. 著

沈洁元 译

www.byteofpython.info

版本：1.20

A Byte of Python

简明 Python 教程

概要

无论您刚接触电脑还是一个有经验的程序员，本书都将有助您学习使用Python语言。

前言

目录表

Python语言可能是第一种即简单又功能强大的编程语言。它不仅适合于初学者，也适合于专业人员使用，更加重要的是，用Python编程是一种愉快的事。本身将帮助你学习这个奇妙的语言，并且向你展示如何即快捷又方便地完成任务——真正意义上"为编程问题提供的完美解决方案！"

本书可以作为Python编程语言的一本指南或者教程。它主要是为新手而设计，不过对于有经验的程序员来说，它同样有用。

即便你对计算机的了解只是如何在计算机上保存文本文件，你都可以通过本书学习Python。如果你有编程经验，你也可以使用本书学习Python。

如果你以前有编程经验，那么你将会对Python语言和其他你所钟爱的编程语言之间的区别感兴趣。对此我为你指出了许多这样的区别。顺便提醒你，Python将很快成为你最喜欢的编程语言！

本书的由来

我最初接触Python是当我需要为我的软件钻石写一个方便安装过程的安装程序的时候。我得在Python和Perl语言中选择一个绑定Qt库。我在网上做了一些研究，偶然发现了一篇文章。那是Eric S. Raymond（著名的电脑高手）谈Python如何成为它最喜欢地编程语言的一篇文章。我同时发现PyQt绑定与Perl-Qt相比要出色得多，所以我选择了Python语言。

之后我开始寻找一本关于Python的优秀书籍。我竟然找不到！虽然我找到了一些O'Reilly的书，不过它们不是太贵就是如同一本参考手册而不是一本指南。我最后使用了Python附带的文档，不过它太简略了。那个文档确实很好的给出了Python的概念，不过不够全面。尽管最后我根据我以前得编程经验掌握了那个文档，不过我觉得它完全不适合于新手。

大约在我首次使用Python语言的六个月之后，我安装了那时最新的Red Hat 9.0 Linux。在我玩弄KWord应用程序的时候，我突然想写一点关于Python的东西。很快我就写了30多页，然后我开始认真地想办法把它变成一本完整的书。经过 多次 的改进和重写，它终于成为了一本有用的完整的Python语言学习指南。我把本书贡献给开源软件者们。

本书来自于我个人学习Python的笔记，不过我尽力让它更加适合别人的口味。

在开源精神的鼓舞下，我收到了许多建设性的建议和批评以及来自热心读者的反馈，它们使这本书变得更加出色。

本书目前的状况

本书目前仍然在进一步完善中。许多章节已经频繁地做了修改。然而本书已经十分成熟了，你一定可以很容易地通过它学习Python。如果你觉得本书中有什么错误或者难懂的地方，请告诉我。

本书将来计划增加更多的章节，包括wxPython，Twisted，有可能的话甚至还有Boa Constructor。

本书的官方网站是www.byteofpython.info。你可以在这个网站上在线阅读本书，也可以下载本书的最新版本或给我反馈。

约定条款

本书（原版）依照创作共用约定（署名-非商业作品-保持一致）发布。简单地说，你只要署上我的名字，就可以免费复制、分发和展示本书。未得到我的允许，你禁止把本书用于商业目的。你在修改本书的时候，必须清楚地标明所有做了改动的地方，你发布修改后的作品时也必须遵照与本书相同的约定。

请访问创作公用约定的网站浏览约定全文，或者查看一个简单易懂的约定描述。那里还有一个连环画似的约定条款的解释。

反馈

我尽了很大的力让这本书即生动又尽可能的准确。然而，如果你找到任何不太令你满意的地方或者错误，或者是需要改进的地方，请告诉我以便我改正它们。你可以把它们通过swaroop@byteofpython.info发送给我。

值得思考的一些东西

> 有两种方式构建软件设计：一种是把软件做得很简单以至于明显找不到缺陷；另一种是把它做得很复杂以至于找不到明显的缺陷。
>
> ——C.A.R. Hoare
>
> 获得人生中的成功需要的专注与坚持不懈多过天才与机会。
>
> ——C.W. Wendte

第1章 介绍

目录表

Python语言是少有的一种可以称得上即简单又功能强大的编程语言。你将惊喜地发现Python语言是多么地简单，它注重的是如何解决问题而不是编程语言的语法和结构。

Python的官方介绍是：

> Python是一种简单易学，功能强大的编程语言，它有高效率的高层数据结构，简单而有效地实现面向对象编程。Python简洁的语法和对动态输入的支持，再加上解释性语言的本质，使得它在大多数平台上的许多领域都是一个理想的脚本语言，特别适用于快速的应用程序开发。

我会在下一节里详细地讨论Python的这些特点。

注释 Python语言的创造者Guido van Rossum是根据英国广播公司的节目“蟒蛇飞行马戏”命名这个语言的，并非他本人特别喜欢蛇缠起它们的长身躯碾死动物觅食。

Python的特色

简单

Python是一种代表简单主义思想的语言。阅读一个良好的Python程序就感觉像是在读英语一样，尽管这个英语的要求非常严格！Python的这种伪代码本质是它最大的优点之一。它使你能够专注于解决问题而不是去搞明白语言本身。

易学

就如同你即将看到的一样，Python极其容易上手。前面已经提到了，Python有极其简单的语法。

免费、开源

Python是FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。FLOSS是基于一个团体分享知识的概念。这是为什么Python如此优秀的原因之一——它是由一群希望看到一个更加优秀的Python的人创造并经常改进着的。

高层语言

当你用Python语言编写程序的时候，你无需考虑诸如如何管理你的程序使用的内存一类的底层细节。

可移植性

由于它的开源本质，Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。如果你小心地避免使用依赖于系统的特性，那么你的所有Python程序无需修改就可以在下述任何平台上面运行。

这些平台包括Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE甚至还有PocketPC！

解释性

这一点需要一些解释。

一个用编译性语言比如C或C++写的程序可以从源文件（即C或C++语言）转换到一个你的计算机使用的语言（二进制代码，即0和1）。这个过程通过编译器和不同的标记、选项完成。当你运行你的程序的时候，连接/转载器软件把你的程序从硬盘复制到内存中并且运行。

而Python语言写的程序不需要编译成二进制代码。你可以直接从源代码 运行 程序。在计算机内部，Python解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。事实上，由于你不再需要担心如何编译程序，如何确保连接转载正确的库等等，所有这一切使得使用Python更加简单。由于你只需要把你的Python程序拷贝到另外一台计算机上，它就可以工作了，这也使得你的Python程序更加易于移植。

面向对象

Python即支持面向过程的编程也支持面向对象的编程。在 面向过程 的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在 面向对象 的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如C++和Java相比，Python以一种非常强大又简单的方式实现面向对象编程。

可扩展性

如果你需要你的一段关键代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用C或C++编写，然后在你的Python程序中使用它们。

可嵌入性

你可以把Python嵌入你的C/C++程序，从而向你的程序用户提供脚本功能。

丰富的库

Python标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV文件、密码系统、GUI（图形用户界面）、Tk和其他与系统有关的操作。记住，只要安装了Python，所有这些功能都是可用的。这被称作Python的“功能齐全”理念。

除了标准库以外，还有许多其他高质量的库，如wxPython、Twisted和Python图像库等等。

Python确实是一种十分精彩又强大的语言。它合理地结合了高性能与使得编写程序简单有趣的特色。

为什么不使用Perl？

也许你以前并不知道，Perl是另外一种极其流行的开源解释性编程语言。

如果你曾经尝试过用Perl语言编写一个大程序，你一定会自己回答这个问题。在规模较小的时候，Perl程序是简单的。它可以胜任于小型的应用程序和脚本，“使工作完成”。然而，当你想开始写一些大一点的程序的时候，Perl程序就变得不实用了。我是通过为Yahoo编写大型Perl程序的经验得出这样的总结的！

与Perl相比，Python程序一定会更简单、更清晰、更易于编写，从而也更加易懂、易维护。我确实也很喜欢Perl，用它来做一些日常的各种事情。不过当我要写一个程序的时候，我总是想到使用Python，这对我来说已经成了十分自然的事。Perl已经经历了多次大的修正和改变，遗憾的是，即将发布的Perl 6似乎仍然没有在这个方面做什么改进。

我感到Perl唯一也是十分重要的优势是它庞大的CPAN库——综合Perl存档网络。就如同这个名字所指的意思一样，这是一个巨大的Perl模块集，它大得让人难以置信——你几乎用这些模块在计算机上做任何事情。Perl的模块比Python多的原因之一是Perl拥有更加悠久的历史。或许我会在comp.lang.python上建议把Perl模块移植到Python上的计划。

另外，新的Parrot虚拟机按设计可以运行完全重新设计的Perl 6也可以运行Python和其他解释性语言如Ruby、PHP和Tcl等等。这意味着你将来 或许 可以在Python上使用所有Perl的模块。这将成为两全其美的事——强大的CPAN库与强大的Python语言结合在一起。我们将拭目以待。

程序员的话

读一下像ESR这样的超级电脑高手谈Python的话，你会感到十分有意思：

- **Eric S. Raymond**是《The Cathedral and the Bazaar》的作者、“开放源码”一词的提出人。他说Python已经成为了他最喜爱的编程语言。这篇文章也是促使我第一次接触Python的真正原动力。
- **Bruce Eckel**著名的《Thinking in Java》和《Thinking in C++》的作者。他说没有一种语言比得上Python使他的工作效率如此之高。同时他说Python可能是唯一一种旨在帮助程序员把事情弄得更加简单的语言。请阅读完整的采访以获得更详细的内容。
- **Peter Norvig**是著名的Lisp语言书籍的作者和Google公司的搜索质量主任（感谢Guido van Rossum告诉我这一点）。他说Python始终是Google的主要部分。事实上你看一下Google招聘的网页就可以验证这一点。在那个网页上，Python知识是对软件工程师的一个必需要求。
- **Bruce Perens**是OpenSource.org和UserLinux项目的一位共同创始人。UserLinux旨在创造一个可以被多家发行商支持标准的Linux发行版。Python击败了其它竞争对手如Perl和Ruby成为UserLinux支持的主要编程语言。

第2章 安装Python

目录表

如果你正在使用一个Linux的发行版比如Fedora或者Mandrake或者其他（你的选择），或者一个BSD系统比如FreeBSD，那么你可能已经在你的系统里安装了Python。

要测试你是否已经随着你的Linux包安装了Python，你可以打开一个shell程序（就像konsole或gnome-terminal）然后输入如下所示的命令 `python -V` 。

```
$ python -V
Python 2.3.4
```

注释 `$` 是shell的提示符。根据你的操作系统的设置，它可能与你那个不同，因此我只用 `$` 符号表示提示符。

如果你看见向上面所示的那样一些版本信息，那么你已经安装了Python了。

如果你得到像这样的消息：

```
$ python -V
bash: python: command not found
```

那么你还没有安装Python。这几乎不可能，只是极其偶尔才会遇到。

在这种情况下，你有两种方法在你的系统上安装Python。

- 利用你的操作系统附带的包管理软件安装二进制包，比如Fedora Linux的yum、Mandrake Linux的urpmi、Debian Linux的apt-get、FreeBSD的pkg_add等等。注意，使用这种方法的话，你需要连接因特网。

  你也可以从别的地方下载二进制包然后拷贝到你的PC中安装。

- 你可以从源代码编译Python然后安装。在网站上有编译的指令。

Windows®用户

Windows®用户可以访问Python.org/download，从网站上下载最新的版本（在写本书的时候，最新版本是2.3.4版）。它的大小大约是9.4MB，与其他大多数语言相比是十分紧凑的。安装过程与其他Windows软件类似。

提示 即便安装程序为你提供了不检查 可选 组件的选项，你也不要不作任何检查！有些组件对你很有用，特别是集成开发环境。

有趣的是，大约70%的Python下载是来自Windows用户的。当然，这并不能说明问题，因为几乎所有的Linux用户已经在安装系统的时候默认安装了Python。

在Windows命令行中使用Python 如果你想要从Windows命令行调用Python，那么你需要先正确的设置PATH变量。

对于Windows 2000、XP、2003，点击控制面板->系统->高级->环境变量。在“系统变量”表单中点击叫做**PATH**的变量，然后编辑这个变量，把**;C:\Python23**加到它的结尾。当然，是Python所在的正确目录名。

对于较旧版本的Windows，把下面这行加到文件C:\AUTOEXEC.BAT中：**PATH=%PATH%;C:\Python23**，然后重新启动系统。对于Windows NT，则使用AUTOEXEC.NT文件。

概括

对于Linux系统，很可能你已经在你的系统里安装了Python。否则，你可以通过你的发行版附带的包管理软件安装Python。对于Windows系统，安装Python就是下载安装程序然后双击它那么简单。从现在起，我们将假设你已经在你的系统里安装了Python。

第3章 最初的步骤

目录表

我们将看一下如何用Python编写运行一个传统的“Hello World”程序。通过它，你将学会如何编写、保存和运行Python程序。

有两种使用Python运行你的程序的方式——使用交互式的带提示符的解释器或使用源文件。我们将学习这两种方法。

使用带提示符的解释器

在命令行的shell提示符下键入**python**，启动解释器。现在输入 `print 'Hello World'` ，然后按**Enter**键。你应该可以看到输出的单词<samp>Hello World</samp>。

对于Windows用户，只要你正确的设置了PATH变量，你应该可以从命令行启动解释器。或者你可以选择使用IDLE程序。IDLE是集成开发环境的缩写。点击开始->程序->Python 2.3->IDLE(Python GUI)。Linux用户也可以使用IDLE。

注意，>>>是你键入Python语句的提示符。

```
$ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>>
```

注意，Python会在下一行立即给出你输出！你刚才键入的是一句Python 语句 。我们使用 `print` （不要惊讶）来打印你提供给它的值。这里，我们提供的是文本 `Hello World` ，它被迅速地打印在屏幕上。

如何退出Python提示符 如果你使用的是Linux/BSD shell，那么按**Ctrl-d**退出提示符。如果是在Windows命令行中，则按**Ctrl-z**再按**Enter**。

挑选一个编辑器

在我们开始讲述以源文件方式编写Python程序之前，我们需要一个编辑器来写源文件。挑选一个编辑器确实是极其重要的。你挑选一个编辑器就如同你挑选一辆你将购买的轿车一样。一个好的编辑器会帮助你方便地编写Python程序，使你地编程旅程更加舒适，帮助你更加快捷安全地到达目的地（实现目标）。

对于编辑器的基本要求之一是语法加亮功能，利用这一功能，你的Python程序的不同部分被标以不同的颜色，这样你可以更好 看清楚 你的程序，使它的运行显得形象化。

如果你使用Windows，那么我建议你使用IDLE。IDLE具备语法加亮功能，还有许多其他的功能，比如允许你在IDLE中运行你的程序。特别值得注意的是：不要使用**Notepad**——它是一个糟糕的选择，因为它没有语法加亮功能，而且更加重要的是，它不支持文本缩进。而我们将会看到文本缩进对于我们来说极其重要。一个好的编辑器，比如IDLE（还有VIM）将会自动帮助你做这些事情。

如果你使用Linux/FreeBSD，那么你有很多种选择。如果你是一位有经验的程序员，你一定已经在使用VIM或者Emacs了。勿庸置疑，它们是两个功能最强大的编辑器。使用它们编写你的Python程序，你将从中受益。我个人使用VIM编写我的大多数程序。如果你是一个初学编程的人，那么你可以使用Kate，它也是我最喜欢的编辑器之一。只要你愿意花时间学习使用VIM或Emacs，那么我强烈建议你一定要学习两者之一，因为从长远看来它们对你是极其有帮助的。

如果你还想寻找一下其他可供选择的编辑器，可以看一下详尽的Python编辑器列表，然后作出你的选择。你也可以使用Python的IDE（集成开发环境）。请看一下详尽的支持Python的IDE列表以获得详尽的信息。一旦你开始编写大型的Python程序，IDE确实很有用。

我再一次重申，请选择一个合适的编辑器——它能使编写Python程序变得更加有趣、方便。

使用源文件

现在让我们重新开始编程。当你学习一种新的编程语言的时候，你编写运行的第一个程序通常都是“Hello World”程序，这已经成为一种传统了。在你运行“Hello World”程序的时候，它所做的事只是说声：“Hello World”。正如提出“Hello World”程序的^[1]所说：“它是编程之神的传统咒语，可以帮助你更好的学习语言。”

启动你选择的编辑器，输入下面这段程序，然后把它保存为 `helloworld.py`。

```
#!/usr/bin/python
# Filename : helloworld.py
print 'Hello World'
```

（源文件：code/helloworld.py）

为了运行这个程序，请打开shell（Linux终端或者DOS提示符），然后键入命令**python helloworld.py**。如果你使用IDLE，请使用菜单Edit->Run Script或者使用键盘快捷方式**Ctrl-F5**。输出如下所示。

```
$ python helloworld.py
Hello World
```

如果你得到的输出与上面所示的一样，那么恭喜！——你已经成功地运行了你的第一个Python程序。

万一你得到一个错误，那么请确保你键入的程序 准确无误 ，然后再运行一下程序。注意Python是大小写敏感的，即 `print` 与 `Print` 不一样——注意前一个是小写 `p` 而后一个是大写 `P` 。另外，确保在每一行的开始字符前没有空格或者制表符——我们将在后面讨论为什么这点是重要的。

让我们思考一下这个程序的前两行。它们被称作 注释 ——任何在#符号右面的内容都是注释。注释主要作为提供给程序读者的笔记。

Python至少应当有第一行那样的特殊形式的注释。它被称作 组织行 ——源文件的头两个字符是#!，后面跟着一个程序。这行告诉你的Linux/Unix系统当你 执行 你的程序的时候，它应该运行哪个解释器。这会在下一节做详细解释。注意，你总是可以通过直接在命令行指定解释器，从而在任何平台上运行你的程序。就如同命令**python helloworld.py**一样。

重要 在你的程序中合理地使用注释以解释一些重要的细节——这将有助于你的程序的读者轻松地理解程序在干什么。记住，这个读者可能就是6个月以后的你！

跟在注释之后的是一句Python 语句 ——它只是打印文本“Hello World”。 `print` 实际上是一个操作符，而“Hello World”被称为一个字符串——别担心我们会在后面详细解释这些术语。

^[1]一位最主要的Perl6/Parrot高手，轰动的《开始Perl》一书的作者。

可执行的Python程序

这部分内容只对Linux/Unix用户适用，不过Windows用户可能也对程序的第一行比较好奇。首先我们需要通过**chmod**命令，给程序可执行的许可，然后 运行 程序。

```
$ chmod a+x helloworld.py
$ ./helloworld.py
Hello World
```

chmod命令用来 改变 文件的 模式 ，给系统中所有用户这个源文件的执行许可。然后我们可以直接通过指定源文件的位置来执行程序。我们使用./来指示程序位于当前目录。

为了更加有趣一些，你可以把你的文件名改成仅仅 `helloworld` ，然后运行**./helloworld**。这样，这个程序仍然可以工作，因为系统知道它必须用源文件第一行指定的那个解释器来运行程序。

只要知道程序的确切位置，你现在就可以运行程序了——但是如果你希望你的程序能够从各个位置运行呢？那样的话，你可以把你的程序保存在PATH环境变量中的目录之一。每当你运行任何程序，系统会查找列在PATH环境变量中的各个目录。然后运行那个程序。你只要简单地把这个源文件复制到PATH所列目录之一就可以使你的程序在任何位置都可用了。

```
$ echo $PATH
/opt/mono/bin/:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
$ cp helloworld.py /home/swaroop/bin/helloworld
$ helloworld
Hello World
```

我们能够用**echo**命令来显示PATH变量，用$给变量名加前缀以向shell表示我们需要这个变量的值。我们看到 `/home/swaroop/bin` 是PATH变量中的目录之一。**swaroop**是我的系统中使用的用户名。通常，在你的系统中也会有一个相似的目录。你也可以把你选择的目录添加到PATH变量中去——这可以通过运行**PATH=$PATH:/home/swaroop/mydir**完成，其中“ `/home/swaroop/mydir` ”是我想要添加到PATH变量中的目录。

当你想要在任何时间、任何地方运行你的程序的时候，这个方法十分有用。它就好像创造你自己的指令，如同**cd**或其他Linux终端或DOS提示符命令那样。

提示 对于Python来说，程序、脚本或者软件都是指同一个东西。

获取帮助

如果你需要某个Python函数或语句的快速信息帮助，那么你可以使用内建的 `help` 功能。尤其在你使用带提示符的命令行的时候，它十分有用。比如，运行 `help(str)` ——这会显示 `str` 类的帮助。str类用于保存你的程序使用的各种文本（字符串）。类将在后面面向对象编程的章节详细解释。

注释 按**q**退出帮助。

类似地，你可以获取Python中几乎所有东西的信息。使用 `help()` 去学习更多关于 `help` 本身的东西！

如果你想要获取关于如 `print` 那样操作符的帮助，那么你需要正确的设置PYTHONDOCS环境变量。这可以在Linux/Unix中轻松地通过**env**命令完成。

```
$ env PYTHONDOCS=/usr/share/doc/python-docs-2.3.4/html/ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help('print')
```

你应该注意到我特意在“ `print` ”上使用了引号，那样Python就可以理解我是希望获取关于“print”的帮助而不是想要它打印东西。

注意，我使用的位置是在Fedora Core 3 Linux中的位置——它可能在不同的发行版和版本中有所不同。

概括

你现在应该可以方便地编写、保存和运行Python程序了。既然你是一个Python用户，让我们学习更多Python的概念。

第4章 基本概念

目录表

仅仅打印“Hello World”就足够了吗？你应该想要做更多的事——你想要得到一些输入，然后做操作，再从中得到一些输出。在Python中，我们可以使用常量和变量来完成这些工作。

一个字面意义上的常量的例子是如同`5`、`1.23`、`9.25e-3`这样的数，或者如同`'This is a string'`、`"It's a string!"`这样的字符串。它们被称作字面意义上的，因为它们具备 字面 的意义——你按照它们的字面意义使用它们的值。数`2`总是代表它自己，而不会是别的什么东西——它是一个常量，因为不能改变它的值。因此，所有这些都被称为字面意义上的常量。

数

在Python中有4种类型的数——整数、长整数、浮点数和复数。

- `2` 是一个整数的例子。
- 长整数不过是大一些的整数。
- `3.23` 和 `52.3E-4` 是浮点数的例子。E标记表示10的幂。在这里，`52.3E-4` 表示 `52.3 * 10` ⁻⁴。
- `(-5+4j)` 和 `(2.3-4.6j)` 是复数的例子。

字符串

字符串是 字符的序列 。字符串基本上就是一组单词。

我几乎可以保证你在每个Python程序中都要用到字符串，所以请特别留心下面这部分的内容。下面告诉你如何在Python中使用字符串。

- 使用单引号（'）

  你可以用单引号指示字符串，就如同 `'Quote me on this'` 这样。所有的空白，即空格和制表符都照原样保留。

- 使用双引号（"）

  在双引号中的字符串与单引号中的字符串的使用完全相同，例如 `"What's your name?"` 。

- 使用三引号（'''或"""）

  利用三引号，你可以指示一个多行的字符串。你可以在三引号中自由的使用单引号和双引号。例如：

  ```
  '''This is a multi-line string. This is the first line. This is the second line. "What
  ```

- 转义符

  假设你想要在一个字符串中包含一个单引号（'），那么你该怎么指示这个字符串？例如，这个字符串是 `What's your name?` 。你肯定不会用 `'What's your name?'` 来指示它，因为Python会弄不明白这个字符串从何处开始，何处结束。所以，你需要指明单引号而不是字符串的结尾。可以通过 转义符 来完成这个任务。你用 `\'` 来指示单引号——注意这个反斜杠。现在你可以把字符串表示为 `'What\'s your name?'` 。

  另一个表示这个特别的字符串的方法是 `"What's your name?"` ，即用双引号。类似地，要在双引号字符串中使用双引号本身的时候，也可以借助于转义符。另外，你可以用转义符 `\\` 来指示反斜杠本身。

  值得注意的一件事是，在一个字符串中，行末的单独一个反斜杠表示字符串在下一行继续，而不是开始一个新的行。例如：

  ```
  "This is the first sentence.\
  This is the second sentence."
  ```

  等价于 `"This is the first sentence. This is the second sentence."`

- 自然字符串

如果你想要指示某些不需要如转义符那样的特别处理的字符串，那么你需要指定一个自然字符串。自然字符串通过给字符串加上前缀 `r` 或 `R` 来指定。例如 `r"Newlines are indicated by \n"` 。

- **Unicode**字符串

  Unicode是书写国际文本的标准方法。如果你想要用你的母语如北印度语或阿拉伯语写文本，那么你需要有一个支持Unicode的编辑器。类似地，Python允许你处理Unicode文本——你只需要在字符串前加上前缀 `u` 或 `U` 。例如， `u"This is a Unicode string."` 。

  记住，在你处理文本文件的时候使用Unicode字符串，特别是当你知道这个文件含有用非英语的语言写的文本。

- 字符串是不可变的

  这意味着一旦你创造了一个字符串，你就不能再改变它了。虽然这看起来像是一件坏事，但实际上它不是。我们将会在后面的程序中看到为什么我们说它不是一个缺点。

- 按字面意义级连字符串

  如果你把两个字符串按字面意义相邻放着，他们会被Python自动级连。例如， `'What\'s' 'your name?'` 会被自动转为 `"What's your name?"` 。

给C/C++程序员的注释 在Python中没有专门的 `char` 数据类型。确实没有需要有这个类型，我相信你不会为此而烦恼。

给Perl/PHP程序员的注释 记住，单引号和双引号字符串是完全相同的——它们没有在任何方面有不同。

给正则表达式用户的注释 一定要用自然字符串处理正则表达式。否则会需要使用很多的反斜杠。例如，后向引用符可以写成 `'\\1'` 或 `r'\1'` 。

变量

仅仅使用字面意义上的常量很快就会引发烦恼——我们需要一种既可以储存信息 又可以对它们进行操作的方法。这是为什么要引入 变量 。变量就是我们想要的东西——它们的值可以变化，即你可以使用变量存储任何东西。变量只是你的计算机中存储信息的一部分内存。与字面意义上的常量不同，你需要一些能够访问这些变量的方法，因此你给变量名字。

标识符的命名

变量是标识符的例子。 标识符 是用来标识 某样东西 的名字。在命名标识符的时候，你要遵循这些规则：

- 标识符的第一个字符必须是字母表中的字母（大写或小写）或者一个下划线（‘ _ ’）。
- 标识符名称的其他部分可以由字母（大写或小写）、下划线（‘ _ ’）或数字（0-9）组成。
- 标识符名称是对大小写敏感的。例如， `myname` 和 `myName` 不是一个标识符。注意前者中的小写n和后者中的大写N。
- 有效 标识符名称的例子有 `i` 、 `__my_name` 、 `name_23` 和 `a1b2_c3` 。
- 无效 标识符名称的例子有 `2things` 、 `this is spaced out` 和 `my-name` 。

数据类型

变量可以处理不同类型的值，称为数据类型。基本的类型是数和字符串，我们已经讨论过它们了。在后面的章节里面，我们会研究怎么用类创造我们自己的类型。

对象

记住，Python把在程序中用到的任何东西都称为 对象 。这是从广义上说的。因此我们不会说“某某 东西 ”，我们说“某个 对象 ”。

给面向对象编程用户的注释 就每一个东西包括数、字符串甚至函数都是对象这一点来说，Python是极其完全地面向对象的。

我们将看一下如何使用变量和字面意义上的常量。保存下面这个例子，然后运行程序。

如何编写Python程序 下面是保存和运行Python程序的标准流程。 1. 打开你最喜欢的编辑器。 2. 输入例子中的程序代码。 3. 用注释中给出的文件名把它保存为一个文件。我按照惯例把所有的Python程序都以扩展名 `.py` 保存。 4. 运行解释器命令**python program.py**或者使用IDLE运行程序。你也可以使用先前介绍的可执行的方法。

```
# Filename : var.py
i = 5
print i
i = i + 1
print i

s = '''This is a multi-line string.
This is the second line.'''
print s
```

（源文件：code/var.py）

```
$ python var.py
5
6
This is a multi-line string.
This is the second line.
```

下面来说明一下这个程序如何工作。首先我们使用赋值运算符（ `=` ）把一个字面意义上的常数 `5` 赋给变量 `i` 。这一行称为一个语句。语句声明需要做某件事情，在这个地方我们把变量名 `i` 与值 `5` 连接在一起。接下来，我们用 `print` 语句打印 `i` 的值，就是把变量的值打印在屏幕上。

然后我们对 `i` 中存储的值加 `1` ，再把它存回 `i` 。我们打印它时，得到期望的值 `6` 。

类似地，我们把一个字面意义上的字符串赋给变量 `s` 然后打印它。

给C/C++程序员的注释 使用变量时只需要给它们赋一个值。不需要声明或定义数据类型。

逻辑行与物理行

物理行是你在编写程序时所 看见 的。逻辑行是Python 看见 的单个语句。Python假定每个 物理行 对应一个 逻辑行 。

逻辑行的例子如 `print 'Hello World'` 这样的语句——如果它本身就是一行（就像你在编辑器中看到的那样），那么它也是一个物理行。

默认地，Python希望每行都只使用一个语句，这样使得代码更加易读。

如果你想要在一个物理行中使用多于一个逻辑行，那么你需要使用分号（;）来特别地标明这种用法。分号表示一个逻辑行/语句的结束。例如：

```
i = 5
print i
```

与下面这个相同：

```
i = 5;
print i;
```

同样也可以写成：

```
i = 5; print i;
```

甚至可以写成：

```
i = 5; print i
```

然而，我**强**烈建议你坚持在每个物理行只写一句逻辑行。仅仅当逻辑行太长的时候，在多于一个物理行写一个逻辑行。这些都是为了尽可能避免使用分号，从而让代码更加易读。事实上，我 从来没有 在Python程序中使用过或看到过分号。

下面是一个在多个物理行中写一个逻辑行的例子。它被称为明确的行连接。

```
s = 'This is a string. \
This continues the string.'
print s
```

它的输出：

```
This is a string. This continues the string.
```

类似地,

```
print \
i
```

与如下写法效果相同：

```
print i
```

有时候，有一种暗示的假设，可以使你不需要使用反斜杠。这种情况出现在逻辑行中使用了圆括号、方括号或波形括号的时候。这被称为暗示的行连接。你会在后面介绍如何使用列表的章节中看到这种用法。

缩进

空白在Python中是重要的。事实上行首的空白是重要的。它称为缩进。在逻辑行首的空白（空格和制表符）用来决定逻辑行的缩进层次，从而用来决定语句的分组。

这意味着同一层次的语句必须有相同的缩进。每一组这样的语句称为一个块。我们将在后面的章节中看到有关块的用处的例子。

你需要记住的一样东西是错误的缩进会引发错误。例如：

```
i = 5
 print 'Value is', i # Error! Notice a single space at the start of the line
print 'I repeat, the value is', i
```

当你运行这个程序的时候，你会得到下面的错误：

```
  File "whitespace.py", line 4
    print 'Value is', i # Error! Notice a single space at the start of the line
    ^
SyntaxError: invalid syntax
```

注意，在第二行的行首有一个空格。Python指示的这个错误告诉我们程序的语法是无效的，即程序没有正确地编写。它告诉你， 你不能随意地开始新的语句块 （当然除了你一直在使用的主块）。何时你能够使用新块，将会在后面的章节，如控制流中详细介绍。

如何缩进 不要混合使用制表符和空格来缩进，因为这在跨越不同的平台的时候，无法正常工作。我 强烈建议 你在每个缩进层次使用 单个制表符 或 两个或四个空格 。 选择这三种缩进风格之一。更加重要的是，选择一种风格，然后一贯地使用它，即 只 使用这一种风格。

概括

现在我们已经学习了很多详细的内容，我们可以开始学习更加令你感兴趣的东西，比如控制流语句。在继续学习之前，请确信你对本章的内容清楚明了。

第5章 运算符与表达式

目录表

你编写的大多数语句（逻辑行）都包含表达式。一个简单的表达式例子如 `2 + 3` 。一个表达式可以分解为运算符和操作数。

运算符 的功能是完成某件事，它们由如 `+` 这样的符号或者其他特定的关键字表示。运算符需要数据来进行运算，这样的数据被称为 操作数 。在这个例子中， `2` 和 `3` 是操作数。

运算符

我们将简单浏览一下运算符和它们的用法：

技巧

你可以交互地使用解释器来计算例子中给出的表达式。例如，为了测试表达式 `2 + 3`，使用交互式的带提示符的Python解释器：

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

运算符	名称	说明	例子
+	加	两个对象相加	3 + 5得到8。'a' + 'b'得到'ab'。
-	减	得到负数或是一个数减去另一个数	-5.2得到一个负数。50 - 24得到26。
*	乘	两个数相乘或是返回一个被重复若干次的字符串	2 *3得到6。'la'* 3得到'lalala'。
**	幂	返回x的y次幂	3 * *4得到81*（即*3 3 3* 3）
/	除	x除以y	4/3得到1（整数的除法得到整数结果）。4.0/3或4/3.0得到1.3333333333333333
//	取整除	返回商的整数部分	4 // 3.0得到1.0
%	取模	返回除法的余数	8%3得到2。-25.5%2.25得到1.5
<<	左移	把一个数的比特向左移一定数目（每个数在内存中都表示为比特或二进制数字，即0和1）	2 << 2得到8。——2按比特表示为10
>>	右移	把一个数的比特向右移一定数目	11 >> 1得到5。——11按比特表示为1011，向右移动1比特后得到101，即十进制的5。
&	按位与	数的按位与	5 & 3得到1。

|	或	数的按位或	5 | 3得到7。
^	按位异或	数的按位异或	5 ^ 3得到6
~	按位翻转	x的按位翻转是-(x+1)	~5得到6。
<	小于	返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	5 < 3返回0（即False）而3 < 5返回1（即True）。比较可以被任意连接：3 < 5 < 7返回True。
>	大于	返回x是否大于y	5 > 3返回True。如果两个操作数都是数字，它们首先被转换为一个共同的类型。否则，它总是返回False。
<=	小于等于	返回x是否小于等于y	x = 3; y = 6; x <= y返回True。
>=	大于等于	返回x是否大于等于y	x = 4; y = 3; x >= y返回True。
==	等于	比较对象是否相等	x = 2; y = 2; x == y返回True。x = 'str'; y = 'stR'; x == y返回False。x = 'str'; y = 'str'; x == y返回True。
!=	不等于	比较两个对象是否不相等	x = 2; y = 3; x != y返回True。
not	布尔“非”	如果x为True，返回False。如果x为False，它返回True。	x = True; not y返回False。
and	布尔“与”	如果x为False，x and y返回False，否则它返回y的计算值。	x = False; y = True; x and y，由于x是False，返回False。在这里，Python不会计算y，因为它知道这个表达式的值肯定是False（因为x是False）。这个现象称为短路计算。
or	布尔“或”	如果x是True，它返回True，否则它返回y的计算值。	x = True; y = False; x or y返回True。短路计算在这里也适用。

运算符优先级

如果你有一个如`2 + 3 * 4`那样的表达式，是先做加法呢，还是先做乘法？我们的中学数学告诉我们应当先做乘法——这意味着乘法运算符的优先级高于加法运算符。

下面这个表给出Python的运算符优先级，从最低的优先级（最松散地结合）到最高的优先级（最紧密地结合）。这意味着在一个表达式中，Python会首先计算表中较下面的运算符，然后在计算列在表上部的运算符。

下面这张表（与Python参考手册中的那个表一模一样）已经顾及了完整的需要。事实上，我建议你使用圆括号来分组运算符和操作数，以便能够明确地指出运算的先后顺序，使程序尽可能地易读。例如，`2 + (3 * 4)`显然比`2 + 3 * 4`清晰。与此同时，圆括号也应该正确使用，而不应该用得过滥（比如`2 + (3 + 4)`）。

运算符	描述
lambda	Lambda表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in，not in	成员测试
is，is not	同一性测试
<，<=，>，>=，!=，==	比较
|	按位或
^	按位异或
&	按位与
<<，>>	移位
+，-	加法与减法
*，/，%	乘法、除法与取余
+x，-x	正负号
~x	按位翻转
**	指数
x.attribute	属性参考
x[index]	下标
x[index:index]	寻址段
f(arguments...)	函数调用
(experession,...)	绑定或元组显示
[expression,...]	列表显示
{key:datum,...}	字典显示
'expression,...'	字符串转换

其中我们还没有接触过的运算符将在后面的章节中介绍。

在表中列在同一行的运算符具有 相同优先级 。例如， `+` 和 `-` 有相同的优先级。

默认地，运算符优先级表决定了哪个运算符在别的运算符之前计算。然而，如果你想要改变它们的计算顺序，你得使用圆括号。例如，你想要在一个表达式中让加法在乘法之前计算，那么你就得写成类似 `(2 + 3) * 4` 的样子。

运算符通常由左向右结合，即具有相同优先级的运算符按照从左向右的顺序计算。例如， `2 + 3 + 4` 被计算成 `(2 + 3) + 4` 。一些如赋值运算符那样的运算符是由右向左结合的，即 `a = b = c` 被处理为 `a = (b = c)` 。

表达式

```
#!/usr/bin/python
# Filename: expression.py

length = 5
breadth = 2
area = length * breadth
print 'Area is', area
print 'Perimeter is', 2 * (length + breadth)
```

（源文件：code/expression.py）

输出

```
$ python expression.py
Area is 10
Perimeter is 14
```

它如何工作

矩形的长度与宽度存储在以它们命名的变量中。我们借助表达式使用它们计算矩形的面积和边长。我们表达式 `length * breadth` 的结果存储在变量 `area` 中，然后用 `print` 语句打印。在另一个打印语句中，我们直接使用表达式 `2 * (length + breadth)` 的值。

另外，注意Python如何打印“漂亮的”输出。尽管我们没有在 `'Area is'` 和变量 `area` 之间指定空格，Python自动在那里放了一个空格，这样我们就可以得到一个清晰漂亮的输出，而程序也变得更加易读（因为我们不需要担心输出之间的空格问题）。这是Python如何使程序员的生活变得更加轻松的一个例子。

概括

我们已经学习了如何使用运算符、操作数和表达式——这些使任何程序的基本组成部分。接下来，我们将学习如何通过语句在我们的程序中使用这些部分。

第6章 控制流

目录表

在到目前为止我们所见到的程序中，总是有一系列的语句，Python忠实地按照它们的顺序执行它们。如果你想要改变语句流的执行顺序，该怎么办呢？例如，你想要让程序做一些决定，根据不同的情况做不同的事情，例如根据时间打印“早上好”或者“晚上好”。

你可能已经猜到了，这是通过控制流语句实现的。在Python中有三种控制流语句——`if`、`for`和`while`。

if语句

`if` 语句用来检验一个条件， 如果 条件为真，我们运行一块语句（称为 if-块 ）， 否则 我们处理另外一块语句（称为 else-块 ）。 else 从句是可选的。

```
#!/usr/bin/python
# Filename: if.py

number = 23
guess = int(raw_input('Enter an integer : '))

if guess == number:
    print 'Congratulations, you guessed it.' # New block starts here
    print "(but you do not win any prizes!)" # New block ends here
elif guess &lt; number:
    print 'No, it is a little higher than that' # Another block
    # You can do whatever you want in a block ...
else:
    print 'No, it is a little lower than that'
    # you must have guess &gt; number to reach here

print 'Done'
# This last statement is always executed, after the if statement is executed
```

（源文件：code/if.py）

输出

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done
$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done
$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

在这个程序中，我们从用户处得到猜测的数，然后检验这个数是否是我们手中的那个。我们把变量 `number` 设置为我们想要的任何整数，在这个例子中是 `23` 。然后，我们使用 `raw_input()` 函数取得用户猜测的数字。函数只是重用的程序段。我们将在下一章学习更多关于函数的知识。

我们为内建的 `raw_input` 函数提供一个字符串，这个字符串被打印在屏幕上，然后等待用户的输入。一旦我们输入一些东西，然后按回车键之后，函数返回输入。对于 `raw_input` 函数来说是一个字符串。我们通过 `int` 把这个字符串转换为整数，并把它存储在变量 `guess` 中。

事实上，`int`是一个类，不过你想在对它所需了解的只是它把一个字符串转换为一个整数（假设这个字符串含有一个有效的整数文本信息）。

接下来，我们将用户的猜测与我们选择的数做比较。如果他们相等，我们打印一个成功的消息。注意我们使用了缩进层次来告诉Python每个语句分别属于哪一个块。这就是为什么缩进在Python如此重要的原因。我希望你能够坚持“每个缩进层一个制表符”的规则。你是这样的吗？

注意`if`语句在结尾处包含一个冒号——我们通过它告诉Python下面跟着一个语句块。

然后，我们检验猜测是否小于我们的数，如果是这样的，我们告诉用户它的猜测大了一点。我们在这里使用的是`elif`从句，它事实上把两个相关联的`if else-if else`语句合并为一个`if-elif-else`语句。这使得程序更加简单，并且减少了所需的缩进数量。

`elif`和`else`从句都必须在逻辑行结尾处有一个冒号，下面跟着一个相应的语句块（当然还包括正确的缩进）。

你也可以在一个if块中使用另外一个`if`语句，等等——这被称为嵌套的`if`语句。

记住，`elif`和`else`部分是可选的。一个最简单的有效`if`语句是：

```
if True:
    print 'Yes, it is true'
```

在Python执行完一个完整的`if`语句以及与它相关联的`elif`和`else`从句之后，它移向`if`语句块的下一个语句。在这个例子中，这个语句块是主块。程序从主块开始执行，而下一个语句是`print 'Done'`语句。在这之后，Python看到程序的结尾，简单的结束运行。

尽管这是一个非常简单的程序，但是我已经在这个简单的程序中指出了许多你应该注意的地方。所有这些都是十分直接了当的（对于那些拥有C/C++背景的用户来说是尤为简单的）。它们在开始时会引起你的注意，但是以后你会对它们感到熟悉、“自然”。

给C/C++程序员的注释 在Python中没有`switch`语句。你可以使用`if..elif..else`语句来完成同样的工作（在某些场合，使用字典会更加快捷。）

while语句

只要在一个条件为真的情况下， `while` 语句允许你重复执行一块语句。 `while` 语句是所谓 循环 语句的一个例子。 `while` 语句有一个可选的 `else` 从句。

```
#!/usr/bin/python
# Filename: while.py

number = 23
running = True

while running:
    guess = int(raw_input('Enter an integer : '))

    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False # this causes the while loop to stop
    elif guess &lt; number:
        print 'No, it is a little higher than that'
    else:
        print 'No, it is a little lower than that'
else:
    print 'The while loop is over.'
    # Do anything else you want to do here

print 'Done'
```

（源文件：code/while.py）

输出

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

它如何工作

在这个程序中，我们仍然使用了猜数游戏作为例子，但是这个例子的优势在于用户可以不断的猜数，直到他猜对为止——这样就不需要像前面那个例子那样为每次猜测重复执行一遍程序。这个例子恰当地说明了 `while` 语句的使用。

我们把 `raw_input` 和 `if` 语句移到了 `while` 循环内，并且在while循环开始前把 `running` 变量设置为 `True` 。首先，我们检验变量 `running` 是否为 `True` ，然后执行后面的 while-块 。在执行了这块程序之后，再次检验条件，在这个例子中，条件是 `running` 变量。如果它是真的，

我们再次执行while-块，否则，我们继续执行可选的else-块，并接着执行下一个语句。

当 `while` 循环条件变为 `False` 的时候， `else` 块才被执行——这甚至也可能是在条件第一次被检验的时候。如果 `while` 循环有一个 `else` 从句，它将始终被执行，除非你的 `while` 循环将永远循环下去不会结束！

`True` 和 `False` 被称为布尔类型。你可以分别把它们等效地理解为值 `1` 和 `0` 。在检验重要条件的时候，布尔类型十分重要，它们并不是真实的值 `1` 。

else块事实上是多余的，因为你可以把其中的语句放在同一块（与 `while` 相同）中，跟在 `while` 语句之后，这样可以取得相同的效果。

给C/C++程序员的注释 记住，你可以在 `while` 循环中使用一个 `else` 从句。

for循环

`for..in` 是另外一个循环语句，它在一序列的对象上 递归 即逐一使用队列中的每个项目。我们会在后面的章节中更加详细地学习序列。

```
#!/usr/bin/python
# Filename: for.py

for i in range(1, 5):
    print i
else:
    print 'The for loop is over'
```

输出

```
$ python for.py
1
2
3
4
The for loop is over
```

它如何工作

在这个程序中，我们打印了一个 序列 的数。我们使用内建的 `range` 函数生成这个数的序列。

我们所做的只是提供两个数， `range` 返回一个序列的数。这个序列从第一个数开始到第二个数为止。例如， `range(1,5)` 给出序列 `[1, 2, 3, 4]` 。默认地， `range` 的步长为1。如果我们为 `range` 提供第三个数，那么它将成为步长。例如， `range(1,5,2)` 给出 `[1,3]` 。记住，range 向上 延伸到第二个数，即它不包含第二个数。

`for` 循环在这个范围内递归—— `for i in range(1,5)` 等价于 `for i in [1, 2, 3, 4]` ，这就如同把序列中的每个数（或对象）赋值给i，一次一个，然后以每个 `i` 的值执行这个程序块。在这个例子中，我们只是打印i的值。

记住， `else` 部分是可选的。如果包含else，它总是在 `for` 循环结束后执行一次，除非遇到break语句。

记住， `for..in` 循环对于任何序列都适用。这里我们使用的是一个由内建 `range` 函数生成的数的列表，但是广义说来我们可以使用任何种类的由任何对象组成的序列！我们会在后面的章节中详细探索这个观点。

给C/C++/Java/C#程序员的注释 Python的 `for` 循环从根本上不同于C/C++的 `for` 循环。C#程序员会注意到Python的 `for` 循环与C#中的 `foreach` 循环十分类似。Java程序员会注意到它与Java 1.5中的 `for (int i : IntArray)` 相似。 在C/C++中，如果你想要写 `for (int i = 0; i < 5; i++)` ，那么用Python，你写成 `for i in range(0,5)` 。你会注意到，Python的 `for` 循环更加简单、明白、不易出错。

break语句

`break` 语句是用来 终止 循环语句的，即哪怕循环条件没有称为 `False` 或序列还没有被完全递归，也停止执行循环语句。

一个重要的注释是，如果你从 `for` 或 `while` 循环中 终止 ，任何对应的循环 `else` 块将不执行。

```
#!/usr/bin/python
# Filename: break.py

while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    print 'Length of the string is', len(s)
print 'Done'
```

（源文件：code/break.py）

输出

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something :       use Python!
Length of the string is 12
Enter something : quit
Done
```

它如何工作

在这个程序中，我们反复地取得用户地输入，然后打印每次输入地长度。我们提供了一个特别的条件来停止程序，即检验用户的输入是否是 `'quit'` 。通过 终止 循环到达程序结尾来停止程序。

输入字符串的长度通过内建的 `len` 函数取得。

记住， `break` 语句也可以在 `for` 循环中使用。

G2的Python诗

我在这里输入的是我所写的一段小诗，称为**G2**的**Python**诗：

```
Programming is fun
When the work is done
if you wanna make your work also fun:
      use Python!
```

continue语句

`continue` 语句被用来告诉Python跳过当前循环块中的剩余语句，然后 继续 进行下一轮循环。

```
#!/usr/bin/python
# Filename: continue.py

while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    if len(s) &lt; 3:
        continue
    print 'Input is of sufficient length'
    # Do other kinds of processing here...
```

（源文件：code/continue.py）

输出

```
$ python continue.py
Enter something : a
Enter something : 12
Enter something : abc
Input is of sufficient length
Enter something : quit
```

它如何工作

在这个程序中，我们从用户处取得输入，但是我们仅仅当它们有至少3个字符长的时候才处理它们。所以，我们使用内建的 `len` 函数来取得长度。如果长度小于3，我们将使用 `continue` 语句忽略块中的剩余的语句。否则，这个循环中的剩余语句将被执行，我们可以在这里做我们希望的任何处理。

注意， `continue` 语句对于 `for` 循环也有效。

概括

我们已经学习了如何使用三种控制流语句—— `if` 、 `while` 和 `for` 以及与它们相关的 `break` 和 `continue` 语句。它们是Python中最常用的部分，熟悉这些控制流是应当掌握的基本技能。

接下来，我们将学习如何创建和使用函数。

第7章 函数

目录表

函数是重用的程序段。它们允许你给一块语句一个名称，然后你可以在你的程序的任何地方使用这个名称任意多次地运行这个语句块。这被称为 调用 函数。我们已经使用了许多内建的函数，比如 `len` 和 `range` 。

函数通过 `def` 关键字定义。def关键字后跟一个函数的 标识符 名称，然后跟一对圆括号。圆括号之中可以包括一些变量名，该行以冒号结尾。接下来是一块语句，它们是函数体。下面这个例子将说明这事实上是十分简单的：

```
#!/usr/bin/python
# Filename: function1.py

def sayHello():
    print 'Hello World!' # block belonging to the function

sayHello() # call the function
```

（源文件：code/function1.py）

输出

```
$ python function1.py
Hello World!
```

它如何工作

我们使用上面解释的语法定义了一个称为 `sayHello` 的函数。这个函数不使用任何参数，因此在圆括号中没有声明任何变量。参数对于函数而言，只是给函数的输入，以便于我们可以传递不同的值给函数，然后得到相应的结果。

函数形参

函数取得的参数是你提供给函数的值，这样函数就可以利用这些值 做 一些事情。这些参数就像变量一样，只不过它们的值是在我们调用函数的时候定义的，而非在函数本身内赋值。

参数在函数定义的圆括号对内指定，用逗号分割。当我们调用函数的时候，我们以同样的方式提供值。注意我们使用过的术语——函数中的参数名称为 形参 而你提供给函数调用的值称为 实参 。

```
#!/usr/bin/python
# Filename: func_param.py

def printMax(a, b):
    if a &gt; b:
        print a, 'is maximum'
    else:
        print b, 'is maximum'

printMax(3, 4) # directly give literal values

x = 5
y = 7

printMax(x, y) # give variables as arguments
```

（源文件：code/func_param.py）

输出

```
$ python func_param.py
4 is maximum
7 is maximum
```

它如何工作

这里，我们定义了一个称为 `printMax` 的函数，这个函数需要两个形参，叫做 `a` 和 `b` 。我们使用 `if..else` 语句找出两者之中较大的一个数，并且打印较大的那个数。

在第一个 `printMax` 使用中，我们直接把数，即实参，提供给函数。在第二个使用中，我们使用变量调用函数。 `printMax(x, y)` 使实参 `x` 的值赋给形参 `a` ，实参 `y` 的值赋给形参 `b` 。在两次调用中，printMax函数的工作完全相同。

局部变量

当你在函数定义内声明变量的时候，它们与函数外具有相同名称的其他变量没有任何关系，即变量名称对于函数来说是 局部 的。这称为变量的 作用域 。所有变量的作用域是它们被定义的块，从它们的名称被定义的那点开始。

```
#!/usr/bin/python
# Filename: func_local.py

def func(x):
    print 'x is', x
    x = 2
    print 'Changed local x to', x

x = 50
func(x)
print 'x is still', x
```

（源文件：code/func_local.py）

输出

```
$ python func_local.py
x is 50
Changed local x to 2
x is still 50
```

它如何工作

在函数中，我们第一次使用 `x` 的 值 的时候，Python使用函数声明的形参的值。

接下来，我们把值 `2` 赋给 `x` 。 `x` 是函数的局部变量。所以，当我们在函数内改变 `x` 的值的时候，在主块中定义的 `x` 不受影响。

在最后一个 `print` 语句中，我们证明了主块中的 `x` 的值确实没有受到影响。

如果你想要为一个定义在函数外的变量赋值，那么你就得告诉Python这个变量名不是局部的，而是 全局 的。我们使用 `global` 语句完成这一功能。没有 `global` 语句，是不可能为定义在函数外的变量赋值的。

你可以使用定义在函数外的变量的值（假设在函数内没有同名的变量）。然而，我并不鼓励你这样做，并且你应该尽量避免这样做，因为这使得程序的读者会不清楚这个变量是在哪里定义的。使用 `global` 语句可以清楚地表明变量是在外面的块定义的。

```
#!/usr/bin/python
# Filename: func_global.py

def func():
    global x

    print 'x is', x
    x = 2
    print 'Changed local x to', x

x = 50
func()
print 'Value of x is', x
```

（源文件：code/func_global.py）

输出

```
$ python func_global.py
x is 50
Changed global x to 2
Value of x is 2
```

它如何工作

`global` 语句被用来声明 `x` 是全局的——因此，当我们在函数内把值赋给 `x` 的时候，这个变化也反映在我们在主块中使用 `x` 的值的时候。

你可以使用同一个 `global` 语句指定多个全局变量。例如 `global x, y, z` 。

默认参数值

对于一些函数，你可能希望它的一些参数是 可选 的，如果用户不想要为这些参数提供值的话，这些参数就使用默认值。这个功能借助于默认参数值完成。你可以在函数定义的形参名后加上赋值运算符（=）和默认值，从而给形参指定默认参数值。

注意，默认参数值应该是一个参数。更加准确的说，默认参数值应该是不可变的——这会在后面的章节中做详细解释。从现在开始，请记住这一点。

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
    print message * times

say('Hello')
say('World', 5)
```

（源文件：code/func_default.py）

输出

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

它如何工作

名为 `say` 的函数用来打印一个字符串任意所需的次数。如果我们不提供一个值，那么默认地，字符串将只被打印一遍。我们通过给形参 `times` 指定默认参数值 `1` 来实现这一功能。

在第一次使用 `say` 的时候，我们只提供一个字符串，函数只打印一次字符串。在第二次使用 `say` 的时候，我们提供了字符串和参数 `5` ，表明我们想要 说 这个字符串消息5遍。

重要 只有在形参表末尾的那些参数可以有默认参数值，即你不能在声明函数形参的时候，先声明有默认值的形参而后声明没有默认值的形参。 这是因为赋给形参的值是根据位置而赋值的。例如， `def func(a, b=5)` 是有效的，但是 `def func(a=5, b)` 是 无效 的。

关键参数

如果你的某个函数有许多参数，而你只想指定其中的一部分，那么你可以通过命名来为这些参数赋值——这被称作 关键参数 ——我们使用名字（关键字）而不是位置（我们前面所一直使用的方法）来给函数指定实参。

这样做有两个 优势 ——一，由于我们不必担心参数的顺序，使用函数变得更加简单了。二、假设其他参数都有默认值，我们可以只给我们想要的那些参数赋值。

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

（源文件：code/func_key.py）

输出

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

它如何工作

名为 `func` 的函数有一个没有默认值的参数，和两个有默认值的参数。

在第一次使用函数的时候， `func(3, 7)` ，参数 `a` 得到值 `3` ，参数 `b` 得到值 `7` ，而参数 `c` 使用默认值 `10` 。

在第二次使用函数 `func(25, c=24)` 的时候，根据实参的位置变量 `a` 得到值 `25` 。根据命名，即关键参数，参数 `c` 得到值 `24` 。变量 `b` 根据默认值，为 `5` 。

在第三次使用 `func(c=50, a=100)` 的时候，我们使用关键参数来完全指定参数值。注意，尽管函数定义中， `a` 在 `c` 之前定义，我们仍然可以在 `a` 之前指定参数 `c` 的值。

return语句

`return` 语句用来从一个函数 返回 即跳出函数。我们也可选从函数 返回一个值 。

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
    if x &gt; y:
        return x
    else:
        return y

print maximum(2, 3)
```

（源文件：code/func_return.py）

输出

```
$ python func_return.py
3
```

它如何工作

`maximum` 函数返回参数中的最大值，在这里是提供给函数的数。它使用简单的 `if..else` 语句来找出较大的值，然后 返回 那个值。

注意，没有返回值的 `return` 语句等价于 `return None` 。 `None` 是Python中表示没有任何东西的特殊类型。例如，如果一个变量的值为 `None` ，可以表示它没有值。

除非你提供你自己的 `return` 语句，每个函数都在结尾暗含有 `return None` 语句。通过运行 `print someFunction()` ，你可以明白这一点，函数 `someFunction` 没有使用 `return` 语句，如同：

```
def someFunction():
    pass
```

`pass` 语句在Python中表示一个空的语句块。

DocStrings

Python有一个很奇妙的特性，称为 文档字符串 ，它通常被简称为 docstrings 。DocStrings是一个重要的工具，由于它帮助你的程序文档更加简单易懂，你应该尽量使用它。你甚至可以在程序运行的时候，从函数恢复文档字符串！

```
#!/usr/bin/python
# Filename: func_doc.py

def printMax(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    x = int(x) # convert to integers, if possible
    y = int(y)

    if x &gt; y:
        print x, 'is maximum'
    else:
        print y, 'is maximum'

printMax(3, 5)
print printMax.__doc__
```

（源文件：code/func_doc.py）

输出

```
$ python func_doc.py
5 is maximum
Prints the maximum of two numbers.

        The two values must be integers.
```

它如何工作

在函数的第一个逻辑行的字符串是这个函数的 文档字符串 。注意，DocStrings也适用于模块和类，我们会在后面相应的章节学习它们。

文档字符串的惯例是一个多行字符串，它的首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述。 强烈建议 你在你的函数中使用文档字符串时遵循这个惯例。

你可以使用 `__doc__` （注意双下划线）调用 `printMax` 函数的文档字符串属性（属于函数的名称）。请记住Python把 每一样东西 都作为对象，包括这个函数。我们会在后面的类一章学习更多关于对象的知识。

如果你已经在Python中使用过 `help()` ，那么你已经看到过DocStings的使用了！它所做的只是抓取函数的 `__doc__` 属性，然后整洁地展示给你。你可以对上面这个函数尝试一下——只是在你的程序中包括 `help(printMax)` 。记住按**q**退出 `help` 。

自动化工具也可以以同样的方式从你的程序中提取文档。因此，我 强烈建议 你对你所写的任何正式函数编写文档字符串。随你的Python发行版附带的**pydoc**命令，与 `help()` 类似地使用DocStrings。

概括

我们已经学习了函数的很多方面的知识，不过注意还有一些方面我们没有涉及。然而，我们已经覆盖了大多数在日常使用中，你可能用到的Python函数知识。

接下来，我们将学习如何创建和使用Python模块。

第8章 模块

目录表

你已经学习了如何在你的程序中定义一次函数而重用代码。如果你想要在其他程序中重用很多函数，那么你该如何编写程序呢？你可能已经猜到了，答案是使用模块。模块基本上就是一个包含了所有你定义的函数和变量的文件。为了在其他程序中重用模块，模块的文件名必须以 `.py` 为扩展名。

模块可以从其他程序 输入 以便利用它的功能。这也是我们使用Python标准库的方法。首先，我们将学习如何使用标准库模块。

```
#!/usr/bin/python
# Filename: using_sys.py

import sys

print 'The command line arguments are:'
for i in sys.argv:
    print i

print '\n\nThe PYTHONPATH is', sys.path, '\n'
```

（源文件：code/using_sys.py）

输出

```
$ python using_sys.py we are arguments
The command line arguments are:
using_sys.py
we
are
arguments

The PYTHONPATH is ['/home/swaroop/byte/code', '/usr/lib/python23.zip',
'/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',
'/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload',
'/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/gtk-2.0']
```

它如何工作

首先，我们利用 `import` 语句 输入 `sys` 模块。基本上，这句语句告诉Python，我们想要使用这个模块。 `sys` 模块包含了与Python解释器和它的环境有关的函数。

当Python执行 `import sys` 语句的时候，它在 `sys.path` 变量中所列目录中寻找 `sys.py` 模块。如果找到了这个文件，这个模块的主块中的语句将被运行，然后这个模块将能够被你 使用 。注意，初始化过程仅在我们 第一次 输入模块的时候进行。另外，“sys”是“system”的缩写。

`sys` 模块中的 `argv` 变量通过使用点号指明—— `sys.argv` ——这种方法的一个优势是这个名称不会与任何在你的程序中使用的 `argv` 变量冲突。另外，它也清晰地表明了这个名称是 `sys` 模块的一部分。

`sys.argv` 变量是一个字符串的 列表 （列表会在后面的章节详细解释）。特别地， `sys.argv` 包含了 命令行参数 的列表，即使用命令行传递给你的程序的参数。

如果你使用IDE编写运行这些程序，请在菜单里寻找一个指定程序的命令行参数的方法。

这里，当我们执行 `python using_sys.py we are arguments` 的时候，我们使用**python**命令运行 `using_sys.py` 模块，后面跟着的内容被作为参数传递给程序。Python为我们把它存储在 `sys.argv` 变量中。

记住，脚本的名称总是 `sys.argv` 列表的第一个参数。所以，在这里， `'using_sys.py'` 是 `sys.argv[0]` 、 `'we'` 是 `sys.argv[1]` 、 `'are'` 是 `sys.argv[2]` 以及 `'arguments'` 是 `sys.argv[3]` 。注意，Python从0开始计数，而非从1开始。

`sys.path` 包含输入模块的目录名列表。我们可以观察到 `sys.path` 的第一个字符串是空的——这个空的字符串表示当前目录也是 `sys.path` 的一部分，这与 `PYTHONPATH` 环境变量是相同的。这意味着你可以直接输入位于当前目录的模块。否则，你得把你的模块放在 `sys.path` 所列的目录之一。

字节编译的.pyc文件

输入一个模块相对来说是一个比较费时的事情，所以Python做了一些技巧，以便使输入模块更加快一些。一种方法是创建 字节编译的文件， 这些文件以 `.pyc` 作为扩展名。字节编译的文件与Python变换程序的中间状态有关（是否还记得Python如何工作的介绍？）。当你在下次从别的程序输入这个模块的时候， `.pyc` 文件是十分有用的——它会快得多，因为一部分输入模块所需的处理已经完成了。另外，这些字节编译的文件也是与平台无关的。所以，现在你知道了那些 `.pyc` 文件事实上是什么了。

from..import语句

如果你想要直接输入 `argv` 变量到你的程序中（避免在每次使用它时打 `sys.` ），那么你可以使用 `from sys import argv` 语句。如果你想要输入所有 `sys` 模块使用的名字，那么你可以使用 `from sys import *` 语句。这对于所有模块都适用。一般说来，应该避免使用 `from..import` 而使用 `import` 语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

模块的name

每个模块都有一个名称，在模块中可以通过语句来找出模块的名称。这在一个场合特别有用——就如前面所提到的，当一个模块被第一次输入的时候，这个模块的主块将被运行。假如我们只想在程序本身被使用的时候运行主块，而在它被别的模块输入的时候不运行主块，我们该怎么做呢？这可以通过模块的**name**属性完成。

```
#!/usr/bin/python
# Filename: using_name.py

if __name__ == '__main__':
    print 'This program is being run by itself'
else:
    print 'I am being imported from another module'
```

（源文件：code/using_name.py）

输出

```
$ python using_name.py
This program is being run by itself
$ python
>>> import using_name
I am being imported from another module
>>>
```

它如何工作

每个Python模块都有它的`__name__`，如果它是`'__main__'`，这说明这个模块被用户单独运行，我们可以进行相应的恰当操作。

制造你自己的模块

创建你自己的模块是十分简单的，你一直在这样做！每个Python程序也是一个模块。你已经确保它具有 `.py` 扩展名了。下面这个例子将会使它更加清晰。

```
#!/usr/bin/python
# Filename: mymodule.py

def sayhi():
    print 'Hi, this is mymodule speaking.'

version = '0.1'

# End of mymodule.py
```

（源文件：code/mymodule.py）

上面是一个 模块 的例子。你已经看到，它与我们普通的Python程序相比并没有什么特别之处。我们接下来将看看如何在我们别的Python程序中使用这个模块。

记住这个模块应该被放置在我们输入它的程序的同一个目录中，或者在 `sys.path` 所列目录之一。

```
#!/usr/bin/python
# Filename: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Version', mymodule.version
```

（源文件：code/mymodule_demo.py）

输出

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

它如何工作

注意我们使用了相同的点号来使用模块的成员。Python很好地重用了相同的记号来，使我们这些Python程序员不需要不断地学习新的方法。

下面是一个使用 `from..import` 语法的版本。

```
#!/usr/bin/python
# Filename: mymodule_demo2.py

from mymodule import sayhi, version
# Alternative:
# from mymodule import *

sayhi()
print 'Version', version
```

（源文件：code/mymodule_demo2.py）

`mymodule_demo2.py` 的输出与 `mymodule_demo.py` 完全相同。

dir()函数

你可以使用内建的 `dir` 函数来列出模块定义的标识符。标识符有函数、类和变量。

当你为 `dir()` 提供一个模块名的时候，它返回模块定义的名称列表。如果不提供参数，它返回当前模块中定义的名称列表。

```
$ python
>>> import sys
>>> dir(sys) # get list of attributes for sys module
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'exc_clear', 'exc_info', 'exc_type',
'excepthook', 'exec_prefix', 'executable', 'exit', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
'meta_path','modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
>>> dir() # get list of attributes for current module
['__builtins__', '__doc__', '__name__', 'sys']
>>>
>>> a = 5 # create a new variable 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # delete/remove a name
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

它如何工作

首先，我们来看一下在输入的 `sys` 模块上使用 `dir` 。我们看到它包含一个庞大的属性列表。

接下来，我们不给 `dir` 函数传递参数而使用它——默认地，它返回当前模块的属性列表。注意，输入的模块同样是列表的一部分。

为了观察 `dir` 的作用，我们定义一个新的变量 `a` 并且给它赋一个值，然后检验 `dir` ，我们观察到在列表中增加了以上相同的值。我们使用 `del` 语句删除当前模块中的变量/属性，这个变化再一次反映在 `dir` 的输出中。

关于 `del` 的一点注释——这个语句在运行后被用来 删除 一个变量/名称。在这个例子中， `del a` ，你将无法再使用变量 `a` ——它就好像从来没有存在过一样。

概括

模块的用处在于它能为你在别的程序中重用它提供的服务和功能。Python附带的标准库就是这样一组模块的例子。我们已经学习了如何使用这些模块以及如何创造我们自己的模块。

接下来，我们将学习一些有趣的概念，它们称为数据结构。

第9章 数据结构

目录表

数据结构基本上就是——它们是可以处理一些 数据 的 结构 。或者说，它们是用来存储一组相关数据的。

在Python中有三种内建的数据结构——列表、元组和字典。我们将会学习如何使用它们，以及它们如何使编程变得简单。

列表

`list` 是处理一组有序项目的数据结构，即你可以在一个列表中存储一个 序列 的项目。假想你有一个购物列表，上面记载着你要买的东西，你就容易理解列表了。只不过在你的购物表上，可能每样东西都独自占有一行，而在Python中，你在每个项目之间用逗号分割。

列表中的项目应该包括在方括号中，这样Python就知道你是在指明一个列表。一旦你创建了一个列表，你可以添加、删除或是搜索列表中的项目。由于你可以增加或删除项目，我们说列表是 可变的 数据类型，即这种类型是可以被改变的。

尽管我一直推迟讨论对象和类，但是现在对它们做一点解释可以使你更好的理解列表。我们会在相应的章节详细探索这个主题。

列表是使用对象和类的一个例子。当你使用变量 `i` 并给它赋值的时候，比如赋整数 `5` ，你可以认为你创建了一个类（类型） `int` 的对象（实例） `i` 。事实上，你可以看一下 `help(int)` 以更好地理解这一点。

类也有方法，即仅仅为类而定义地函数。仅仅在你有一个该类的对象的时候，你才可以使用这些功能。例如，Python为 `list` 类提供了 `append` 方法，这个方法让你在列表尾添加一个项目。例如 `mylist.append('an item')` 列表 `mylist` 中增加那个字符串。注意，使用点号来使用对象的方法。

一个类也有域，它是仅仅为类而定义的变量。仅仅在你有一个该类的对象的时候，你才可以使用这些变量/名称。类也通过点号使用，例如 `mylist.field` 。

```
#!/usr/bin/python
# Filename: using_list.py

# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print 'I have', len(shoplist),'items to purchase.'

print 'These items are:', # Notice the comma at end of the line
for item in shoplist:
    print item,

print '\nI also have to buy rice.'
shoplist.append('rice')
print 'My shopping list is now', shoplist

print 'I will sort my list now'
shoplist.sort()
print 'Sorted shopping list is', shoplist

print 'The first item I will buy is', shoplist[0]
olditem = shoplist[0]
del shoplist[0]
print 'I bought the', olditem
print 'My shopping list is now', shoplist
```

（源文件：code/using_list.py）

输出

```
$ python using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

它如何工作

变量 `shoplist` 是某人的购物列表。在 `shoplist` 中，我们只存储购买的东西的名字字符串，但是记住，你可以在列表中添加 任何种类的对象 包括数甚至其他列表。

我们也使用了 `for..in` 循环在列表中各项目间递归。从现在开始，你一定已经意识到列表也是一个序列。序列的特性会在后面的章节中讨论。

注意，我们在 `print` 语句的结尾使用了一个 逗号 来消除每个 `print` 语句自动打印的换行符。这样做有点难看，不过确实简单有效。

接下来，我们使用 `append` 方法在列表中添加了一个项目，就如前面已经讨论过的一样。然后我们通过打印列表的内容来检验这个项目是否确实被添加进列表了。打印列表只需简单地把列表传递给 `print` 语句，我们可以得到一个整洁的输出。

再接下来，我们使用列表的 `sort` 方法来对列表排序。需要理解的是，这个方法影响列表本身，而不是返回一个修改后的列表——这与字符串工作的方法不同。这就是我们所说的列表是 可变的 而字符串是 不可变的 。

最后，但我们完成了在市场购买一样东西的时候，我们想要把它从列表中删除。我们使用 `del` 语句来完成这个工作。这里，我们指出我们想要删除列表中的哪个项目，而 `del` 语句为我们从列表中删除它。我们指明我们想要删除列表中的第一个元素，因此我们使用 `del shoplist[0]` （记住，Python从0开始计数）。

如果你想要知道列表对象定义的所有方法，可以通过 `help(list)` 获得完整的知识。

元组

元组和列表十分类似，只不过元组和字符串一样是 不可变的 即你不能修改元组。元组通过圆括号中用逗号分割的项目定义。元组通常用在使语句或用户定义的函数能够安全地采用一组值的时候，即被使用的元组的值不会改变。

```
#!/usr/bin/python
# Filename: using_tuple.py

zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)

new_zoo = ('monkey', 'dolphin', zoo)
print 'Number of animals in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from old zoo are', new_zoo[2]
print 'Last animal brought from old zoo is', new_zoo[2][2]
```

（源文件：code/using_tuple.py）

输出

```
$ python using_tuple.py
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
All animals in new zoo are ('monkey', 'dolphin', ('wolf', 'elephant', 'penguin'))
Animals brought from old zoo are ('wolf', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
```

它如何工作

变量 `zoo` 是一个元组，我们看到 `len` 函数可以用来获取元组的长度。这也表明元组也是一个序列。

由于老动物园关闭了，我们把动物转移到新动物园。因此， `new_zoo` 元组包含了一些已经在那里的动物和从老动物园带过来的动物。回到话题，注意元组之内的元组不会失去它的身份。

我们可以通过一对方括号来指明某个项目的位置从而来访问元组中的项目，就像我们对列表的用法一样。这被称作 索引 运算符。我们使用 `new_zoo[2]` 来访问 `new_zoo` 中的第三个项目。我们使用 `new_zoo[2][2]` 来访问 `new_zoo` 元组的第三个项目的第三个项目。

含有**0**个或**1**个项目的元组。一个空的元组由一对空的圆括号组成，如 `myempty = ()` 。然而，含有单个元素的元组就不那么简单了。你必须在第一个（唯一一个）项目后跟一个逗号，这样Python才能区分元组和表达式中一个带圆括号的对象。即如果你想要的是一个包含项

目 `2` 的元组的时候，你应该指明 `singleton = (2 , )` 。

给Perl程序员的注释 列表之中的列表不会失去它的身份，即列表不会像Perl中那样被打散。同样元组中的元组，或列表中的元组，或元组中的列表等等都是如此。只要是Python，它们就只是使用另一个对象存储的对象。

元组最通常的用法是用在打印语句中，下面是一个例子：

```
#!/usr/bin/python
# Filename: print_tuple.py

age = 22
name = 'Swaroop'

print '%s is %d years old' % (name, age)
print 'Why is %s playing with that python?' % name
```

（源文件：code/print_tuple.py）

输出

```
$ python print_tuple.py
Swaroop is 22 years old
Why is Swaroop playing with that python?
```

它如何工作

`print` 语句可以使用跟着 `%` 符号的项目元组的字符串。这些字符串具备定制的功能。定制让输出满足某种特定的格式。定制可以是 `%s` 表示字符串或 `%d` 表示整数。元组必须按照相同的顺序来对应这些定制。

观察我们使用的第一个元组，我们首先使用 `%s` ，这对应变量 `name` ，它是元组中的第一个项目。而第二个定制是 `%d` ，它对应元组的第二个项目 `age` 。

Python在这里所做的是把元组中的每个项目转换成字符串并且用字符串的值替换定制的位置。因此 `%s` 被替换为变量 `name` 的值，依此类推。

`print` 的这个用法使得编写输出变得极其简单，它避免了许多字符串操作。它也避免了我们一直以来使用的逗号。

在大多数时候，你可以只使用 `%s` 定制，而让Python来提你处理剩余的事情。这种方法对数同样奏效。然而，你可能希望使用正确的定制，从而可以避免多一层的检验程序是否正确。

在第二个 `print` 语句中，我们使用了一个定制，后面跟着 `%` 符号后的单个项目——没有圆括号。这只在字符串中只有一个定制的时候有效。

字典

字典类似于你通过联系人名字查找地址和联系人详细情况的地址簿，即，我们把键（名字）和值（详细情况）联系在一起。注意，键必须是唯一的，就像如果有两个人恰巧同名的话，你无法找到正确的信息。

注意，你只能使用不可变的对象（比如字符串）来作为字典的键，但是你可以不可变或可变的对象作为字典的值。基本说来就是，你应该只使用简单的对象作为键。

键值对在字典中以这样的方式标记：`d = {key1 : value1, key2 : value2 }`。注意它们的键/值对用冒号分割，而各个对用逗号分割，所有这些都包括在花括号中。

记住字典中的键/值对是没有顺序的。如果你想要一个特定的顺序，那么你应该在使用前自己对它们排序。

字典是 `dict` 类的实例/对象。

```
#!/usr/bin/python
# Filename: using_dict.py

# 'ab' is short for 'a'ddress'b'ook

ab = {       'Swaroop'   : 'swaroopch@byteofpython.info',
             'Larry'     : 'larry@wall.org',
             'Matsumoto' : 'matz@ruby-lang.org',
             'Spammer'   : 'spammer@hotmail.com'
     }

print "Swaroop's address is %s" % ab['Swaroop']

# Adding a key/value pair
ab['Guido'] = 'guido@python.org'

# Deleting a key/value pair
del ab['Spammer']

print '\nThere are %d contacts in the address-book\n' % len(ab)
for name, address in ab.items():
    print 'Contact %s at %s' % (name, address)

if 'Guido' in ab: # OR ab.has_key('Guido')
    print "\nGuido's address is %s" % ab['Guido']
```

（源文件：code/using_dict.py）

输出

```
$ python using_dict.py
Swaroop's address is swaroopch@byteofpython.info

There are 4 contacts in the address-book

Contact Swaroop at swaroopch@byteofpython.info
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org
Contact Guido at guido@python.org

Guido's address is guido@python.org
```

它如何工作

我们使用已经介绍过的标记创建了字典 `ab` 。然后我们使用在列表和元组章节中已经讨论过的索引操作符来指定键，从而使用键/值对。我们可以看到字典的语法同样十分简单。

我们可以使用索引操作符来寻址一个键并为它赋值，这样就增加了一个新的键/值对，就像在上面的例子中我们对Guido所做的一样。

我们可以使用我们的老朋友—— `del` 语句来删除键/值对。我们只需要指明字典和用索引操作符指明要删除的键，然后把它们传递给 `del` 语句就可以了。执行这个操作的时候，我们无需知道那个键所对应的值。

接下来，我们使用字典的 `items` 方法，来使用字典中的每个键/值对。这会返回一个元组的列表，其中每个元组都包含一对项目——键与对应的值。我们抓取这个对，然后分别赋给 `for..in` 循环中的变量 `name` 和 `address` 然后在for－块中打印这些值。

我们可以使用 `in` 操作符来检验一个键/值对是否存在，或者使用 `dict` 类的 `has_key` 方法。你可以使用 `help(dict)` 来查看 `dict` 类的完整方法列表。

关键字参数与字典。如果换一个角度看待你在函数中使用的关键字参数的话，你已经使用了字典了！只需想一下——你在函数定义的参数列表中使用的键/值对。当你在函数中使用变量的时候，它只不过是使用一个字典的键（这在编译器设计的术语中被称作 符号表 ）。

序列

列表、元组和字符串都是序列，但是序列是什么，它们为什么如此特别呢？序列的两个主要特点是索引操作符和切片操作符。索引操作符让我们可以从序列中抓取一个特定项目。切片操作符让我们能够获取序列的一个切片，即一部分序列。

```
#!/usr/bin/python
# Filename: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']

# Indexing or 'Subscription' operation
print 'Item 0 is', shoplist[0]
print 'Item 1 is', shoplist[1]
print 'Item 2 is', shoplist[2]
print 'Item 3 is', shoplist[3]
print 'Item -1 is', shoplist[-1]
print 'Item -2 is', shoplist[-2]

# Slicing on a list
print 'Item 1 to 3 is', shoplist[1:3]
print 'Item 2 to end is', shoplist[2:]
print 'Item 1 to -1 is', shoplist[1:-1]
print 'Item start to end is', shoplist[:]

# Slicing on a string
name = 'swaroop'
print 'characters 1 to 3 is', name[1:3]
print 'characters 2 to end is', name[2:]
print 'characters 1 to -1 is', name[1:-1]
print 'characters start to end is', name[:]
```

（源文件：code/seq.py）

输出

```
$ python seq.py
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

它如何工作

首先，我们来学习如何使用索引来取得序列中的单个项目。这也被称作是下标操作。每当你用方括号中的一个数来指定一个序列的时候，Python会为你抓取序列中对应位置的项目。记住，Python从0开始计数。因此，`shoplist[0]`抓取第一个项目，`shoplist[3]`抓取`shoplist`序列中的第四个元素。

索引同样可以是负数，在那样的情况下，位置是从序列尾开始计算的。因此，`shoplist[-1]`表示序列的最后一个元素而`shoplist[-2]`抓取序列的倒数第二个项目。

切片操作符是序列名后跟一个方括号，方括号中有一对可选的数字，并用冒号分割。注意这与你使用的索引操作符十分相似。记住数是可选的，而冒号是必须的。

切片操作符中的第一个数（冒号之前）表示切片开始的位置，第二个数（冒号之后）表示切片到哪里结束。如果不指定第一个数，Python就从序列首开始。如果没有指定第二个数，则Python会停止在序列尾。注意，返回的序列从开始位置 开始 ，刚好在 结束 位置之前结束。即开始位置是包含在序列切片中的，而结束位置被排斥在切片外。

这样，`shoplist[1:3]`返回从位置1开始，包括位置2，但是停止在位置3的一个序列切片，因此返回一个含有两个项目的切片。类似地，`shoplist[:]`返回整个序列的拷贝。

你可以用负数做切片。负数用在从序列尾开始计算的位置。例如，`shoplist[:-1]`会返回除了最后一个项目外包含所有项目的序列切片。

使用Python解释器交互地尝试不同切片指定组合，即在提示符下你能够马上看到结果。序列的神奇之处在于你可以用相同的方法访问元组、列表和字符串。

参考

当你创建一个对象并给它赋一个变量的时候，这个变量仅仅 参考 那个对象，而不是表示这个对象本身！也就是说，变量名指向你计算机中存储那个对象的内存。这被称作名称到对象的绑定。

一般说来，你不需要担心这个，只是在参考上有些细微的效果需要你注意。这会通过下面这个例子加以说明。

```
#!/usr/bin/python
# Filename: reference.py

print 'Simple Assignment'
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist is just another name pointing to the same object!

del shoplist[0]

print 'shoplist is', shoplist
print 'mylist is', mylist
# notice that both shoplist and mylist both print the same list without
# the 'apple' confirming that they point to the same object

print 'Copy by making a full slice'
mylist = shoplist[:] # make a copy by doing a full slice
del mylist[0] # remove first item

print 'shoplist is', shoplist
print 'mylist is', mylist
# notice that now the two lists are different
```

（源文件：code/reference.py）

输出

```
$ python reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

它如何工作

大多数解释已经在程序的注释中了。你需要记住的只是如果你想要复制一个列表或者类似的序列或者其他复杂的对象（不是如整数那样的简单 对象），那么你必须使用切片操作符来取得拷贝。如果你只是想要使用另一个变量名，两个名称都 参考 同一个对象，那么如果你不小心的话，可能会引来各种麻烦。

给Perl程序员的注释 记住列表的赋值语句不创建拷贝。你得使用切片操作符来建立序列的拷贝。

更多字符串的内容

我们已经在前面详细讨论了字符串。我们还需要知道什么呢？那么，你是否知道字符串也是对象，同样具有方法。这些方法可以完成包括检验一部分字符串和去除空格在内的各种工作。

你在程序中使用的字符串都是 `str` 类的对象。这个类的一些有用的方法会在下面这个例子中说明。如果要了解这些方法的完整列表，请参见 `help(str)` 。

```
#!/usr/bin/python
# Filename: str_methods.py

name = 'Swaroop' # This is a string object

if name.startswith('Swa'):
    print 'Yes, the string starts with "Swa"'

if 'a' in name:
    print 'Yes, it contains the string "a"'

if name.find('war') != -1:
    print 'Yes, it contains the string "war"'

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print delimiter.join(mylist)
```

（源文件：code/str_methods.py）

输出

```
$ python str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

它如何工作

这里，我们看到使用了许多字符串方法。 `startwith` 方法是用来测试字符串是否以给定字符串开始。 `in` 操作符用来检验一个给定字符串是否为另一个字符串的一部分。

`find` 方法用来找出给定字符串在另一个字符串中的位置，或者返回-1以表示找不到子字符串。 `str` 类也有以一个作为分隔符的字符串 `join` 序列的项目的整洁的方法，它返回一个生成的大字符串。

概括

我们已经详细探讨了多种Python内建的数据结构。这些数据结构将是编写程序时至关重要的部分。

现在我们已经掌握了很多Python的基本知识，我们接下来将学习如何设计和编写一个实用的Python程序。

第10章 解决问题——编写一个Python脚本

目录表

我们已经研究了Python语言的众多内容，现在我们将来学习一下怎么把这些内容结合起来。我们将设计编写一个能够 做 一些确实有用的事情的程序。

我提出的问题是： 我想要一个可以为我的所有重要文件创建备份的程序。

尽管这是一个简单的问题，但是问题本身并没有给我们足够的信息来解决它。进一步的分析是必需的。例如，我们如何确定该备份哪些文件？备份保存在哪里？我们怎么样存储备份？

在恰当地分析了这个问题之后，我们开始设计我们的程序。我们列了一张表，表示我们的程序应该如何工作。对于这个问题，我已经创建了下面这个列表以说明 我 如何让它工作。如果是你设计的话，你可能不会这样来解决问题——每个人都有其做事的方法，这很正常。

1. 需要备份的文件和目录由一个列表指定。
2. 备份应该保存在主备份目录中。
3. 文件备份成一个zip文件。
4. zip存档的名称是当前的日期和时间。
5. 我们使用标准的**zip**命令，它通常默认地随Linux/Unix发行版提供。Windows用户可以使用Info-Zip程序。注意你可以使用任何地存档命令，只要它有命令行界面就可以了，那样的话我们可以从我们的脚本中传递参数给它。

解决方案

当我们基本完成程序的设计，我们就可以编写代码了，它是对我们的解决方案的实施。

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1\. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or something like

# 2\. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3\. The files are backed up into a zip file.
# 4\. The name of the zip archive is the current date and time
target = target_dir + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5\. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

（源文件：code/backup_ver1.py）

输出

```
$ python backup_ver1.py
Successful backup to /mnt/e/backup/20041208073244.zip
```

现在，我们已经处于测试环节了，在这个环节，我们测试我们的程序是否正确工作。如果它与我们所期望的不一样，我们就得调试我们的程序，即消除程序中的 瑕疵 （错误）。

它如何工作

接下来你将看到我们如何把 设计 一步一步地转换为 代码 。

我们使用了 `os` 和 `time` 模块，所以我们输入它们。然后，我们在 `source` 列表中指定需要备份的文件和目录。目标目录是我们想要存储备份文件的地方，它由 `target_dir` 变量指定。zip 归档的名称是目前的日期和时间，我们使用 `time.strftime()` 函数获得。它还包括 `.zip` 扩展名，将被保存在 `target_dir` 目录中。

`time.strftime()` 函数需要我们在上面的程序中使用的那种定制。 `%Y` 会被无世纪的年份所替代。 `%m` 会被01到12之间的一个十进制月份数替代，其他依次类推。这些定制的详细情况可以在《Python参考手册》中获得。《Python参考手册》包含在你的Python发行版中。注意这些定制与用于 `print` 语句的定制（ `%` 后跟一个元组）类似（但不完全相同）

我们使用加法操作符来 级连 字符串，即把两个字符串连接在一起返回一个新的字符串。通过这种方式，我们创建了目标zip文件的名称。接着我们创建了 `zip_command` 字符串，它包含我们将要执行的命令。你可以在shell（Linux终端或者DOS提示符）中运行它，以检验它是否工作。

zip命令有一些选项和参数。 `-q` 选项用来表示zip命令安静地工作。 `-r` 选项表示zip命令对目录递归地工作，即它包括子目录以及子目录中的文件。两个选项可以组合成缩写形式 `-qr` 。选项后面跟着待创建的zip归档的名称，然后再是待备份的文件和目录列表。我们使用已经学习过的字符串 `join` 方法把 `source` 列表转换为字符串。

最后，我们使用 `os.system` 函数 运行 命令，利用这个函数就好像在 系统 中运行命令一样。即在shell中运行命令——如果命令成功运行，它返回0，否则它返回错误号。

根据命令的输出，我们打印对应的消息，显示备份是否创建成功。好了，就是这样我们已经创建了一个脚本来对我们的重要文件做备份！

给Windows用户的注释 你可以把 `source` 列表和 `target` 目录设置成任何文件和目录名，但是在Windows中你得小心一些。问题是Windows把反斜杠（\）作为目录分隔符，而Python用反斜杠表示转义符！所以，你得使用转义符来表示反斜杠本身或者使用自然字符串。例如，使用 `'C:\\Documents'` 或 `r'C:\Documents'` 而不是 `'C:\Documents'` ——你在使用一个不知名的转义符 `\D` ！

现在我们已经有了一个可以工作的备份脚本，我们可以在任何我们想要建立文件备份的时候使用它。建议Linux/Unix用户使用前面介绍的可执行的方法，这样就可以在任何地方任何时候运行备份脚本了。这被称为软件的实施环节或开发环节。

上面的程序可以正确工作，但是（通常）第一个程序并不是与你所期望的完全一样。例如，可能有些问题你没有设计恰当，又或者你在输入代码的时候发生了一点错误，等等。正常情况下，你应该回到设计环节或者调试程序。

第一个版本的脚本可以工作。然而，我们可以对它做些优化以便让它在我们的日常工作中变得更好。这称为软件的维护环节。

我认为优化之一是采用更好的文件名机制——使用 时间 作为文件名，而当前的 日期 作为目录名，存放在主备份目录中。这样做的一个优势是你的备份会以等级结构存储，因此它就更加容易管理了。另外一个优势是文件名的长度也可以变短。还有一个优势是采用各自独立的文件夹可以帮助你方便地检验你是否在每一天创建了备份，因为只有在你创建了备份，才会出现那天的目录。

```
#!/usr/bin/python
# Filename: backup_ver2.py

import os
import time

# 1\. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or something like

# 2\. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3\. The files are backed up into a zip file.
# 4\. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print 'Successfully created directory', today

# The name of the zip file
target = today + os.sep + now + '.zip'

# 5\. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

（源文件：code/backup_ver2.py）

输出

```
$ python backup_ver2.py
Successfully created directory /mnt/e/backup/20041208
Successful backup to /mnt/e/backup/20041208/080020.zip

$ python backup_ver2.py
Successful backup to /mnt/e/backup/20041208/080428.zip
```

它如何工作

两个程序的大部分是相同的。改变的部分主要是使用 `os.exists` 函数检验在主备份目录中是否有以当前日期作为名称的目录。如果没有，我们使用 `os.mkdir` 函数创建。

注意 `os.sep` 变量的用法——这会根据你的操作系统给出目录分隔符，即在Linux、Unix下它是 `'/'` ，在Windows下它是 `'\\'` ，而在Mac OS下它是 `':'` 。使用 `os.sep` 而非直接使用字符，会使我们的程序具有移植性，可以在上述这些系统下工作。

第二个版本在我做较多备份的时候还工作得不错，但是如果有极多备份的时候，我发现要区分每个备份是干什么的，会变得十分困难！例如，我可能对程序或者演讲稿做了一些重要的改变，于是我想要把这些改变与zip归档的名称联系起来。这可以通过在zip归档名上附带一个用户提供的注释来方便地实现。

```
#!/usr/bin/python
# Filename: backup_ver3.py

import os
import time

# 1\. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or something like

# 2\. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3\. The files are backed up into a zip file.
# 4\. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = raw_input('Enter a comment --&gt; ')
if len(comment) == 0: # check if a comment was entered
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' +
        comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print 'Successfully created directory', today

# 5\. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

（源文件：code/backup_ver3.py）

输出

```
$ python backup_ver3.py
File "backup_ver3.py", line 25
target = today + os.sep + now + '_' +
                                    ^
SyntaxError: invalid syntax
```

它如何（不）工作

这个程序不工作！Python说有一个语法错误，这意味着脚本不满足Python可以识别的结构。当我们观察Python给出的错误的时候，它也告诉了我们它检测出错误的位置。所以我们从那行开始 调试 我们的程序。

通过仔细的观察，我们发现一个逻辑行被分成了两个物理行，但是我们并没有指明这两个物理行属于同一逻辑行。基本上，Python发现加法操作符（+）在那一逻辑行没有任何操作数，因此它不知道该如何继续。记住我们可以使用物理行尾的反斜杠来表示逻辑行在下一物理行继续。所以，我们修正了程序。这被称为修订。

```
#!/usr/bin/python
# Filename: backup_ver4.py

import os
import time

# 1\. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or something like

# 2\. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3\. The files are backed up into a zip file.
# 4\. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = raw_input('Enter a comment --&gt; ')
if len(comment) == 0: # check if a comment was entered
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' + \
        comment.replace(' ', '_') + '.zip'
    # Notice the backslash!

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print 'Successfully created directory', today

# 5\. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

（源文件：code/backup_ver4.py）

输出

```
$ python backup_ver4.py
Enter a comment --&gt; added new examples
Successful backup to /mnt/e/backup/20041208/082156_added_new_examples.zip

$ python backup_ver4.py
Enter a comment --&gt;
Successful backup to /mnt/e/backup/20041208/082316.zip
```

它如何工作

这个程序现在工作了！让我们看一下版本三中作出的实质性改进。我们使用 `raw_input` 函数得到用户的注释，然后通过 `len` 函数找出输入的长度以检验用户是否确实输入了什么东西。如果用户只是按了回车（比如这只是一个惯例备份，没有做什么特别的修改），那么我们就如之前那样继续操作。

然而，如果提供了注释，那么它会被附加到zip归档名，就在 `.zip` 扩展名之前。注意我们把注释中的空格替换成下划线——这是因为处理这样的文件名要容易得多。

对于大多数用户来说，第四个版本是一个满意的工作脚本了，但是它仍然有进一步改进的空间。比如，你可以在程序中包含 交互 程度——你可以用 `-v` 选项来使你的程序更具交互性。

另一个可能的改进是使文件和目录能够通过命令行直接传递给脚本。我们可以通过 `sys.argv` 列表来获取它们，然后我们可以使用 `list` 类提供的 `extend` 方法把它们加到 `source` 列表中去。

我还希望有的一个优化是使用**tar**命令替代**zip**命令。这样做的一个优势是在你结合使用**tar**和**gzip**命令的时候，备份会更快更小。如果你想要在Windows中使用这些归档，WinZip也能方便地处理这些 `.tar.gz` 文件。**tar**命令在大多数Linux/Unix系统中都是默认可用的。Windows用户也可以下载安装它。

命令字符串现在将称为：

```
tar = 'tar -cvzf %s %s -X /home/swaroop/excludes.txt' % (target, ' '.join(srcdir))
```

选项解释如下：

- `-c` 表示创建一个归档。
- `-v` 表示交互，即命令更具交互性。
- `-z` 表示使用**gzip**滤波器。
- `-f` 表示**强**迫创建归档，即如果已经有一个同名文件，它会被替换。
- `-X` 表示含在指定文件名列表中的文件会被排除在备份之外。例如，你可以在文件中指定 `*~`，从而不让备份包括所有以 `~` 结尾的文件。

重要 最理想的创建这些归档的方法是分别使用 `zipfile` 和 `tarfile` 。它们是Python标准库的一部分，可以供你使用。使用这些库就避免了使用 `os.system` 这个不推荐使用的函数，它容易引发严重的错误。 然而，我在本节中使用 `os.system` 的方法来创建备份，这纯粹是为了教学的需要。这样的话，例子就可以简单到让每个人都能够理解，同时也已经足够用了。

软件开发过程

现在，我们已经走过了编写一个软件的各个环节。这些环节可以概括如下：

1. 什么（分析）
2. 如何（设计）
3. 编写（实施）
4. 测试（测试与调试）
5. 使用（实施或开发）
6. 维护（优化）

重要 我们创建这个备份脚本的过程是编写程序的推荐方法——进行分析与设计。开始时实施一个简单的版本。对它进行测试与调试。使用它以确信它如预期那样地工作。再增加任何你想要的特性，根据需要一次次重复这个编写－测试－使用的周期。记住"软件是长出来的，而不是建造的"。

概括

我们已经学习如何创建我们自己的Python程序/脚本，以及在编写这个程序中所设计到的不同的状态。你可以发现它们在创建你自己的程序的时候会十分有用，让你对Python以及解决问题都变得更加得心应手。

接下来，我们将讨论面向对象的编程。

第11章 面向对象的编程

目录表

到目前为止，在我们的程序中，我们都是根据操作数据的函数或语句块来设计程序的。这被称为 面向过程的 编程。还有一种把数据和功能结合起来，用称为对象的东西包裹起来组织程序的方法。这种方法称为 面向对象的 编程理念。在大多数时候你可以使用过程性编程，但是有些时候当你想要编写大型程序或是寻求一个更加合适的解决方案的时候，你就得使用面向对象的编程技术。

类和对象是面向对象编程的两个主要方面。类创建一个新类型，而对象这个类的 实例 。这类似于你有一个 `int` 类型的变量，这存储整数的变量是 `int` 类的实例（对象）。

给C/C++/Java/C#程序员的注释 注意，即便是整数也被作为对象（属于 `int` 类）。这和C++、Java（1.5版之前）把整数纯粹作为类型是不同的。通过 `help(int)` 了解更多这个类的详情。 C#和Java 1.5程序员会熟悉这个概念，因为它类似与 封装与解封装 的概念。

对象可以使用普通的 属于 对象的变量存储数据。属于一个对象或类的变量被称为域。对象也可以使用 属于 类的函数来具有功能。这样的函数被称为类的方法。这些术语帮助我们把它们与孤立的函数和变量区分开来。域和方法可以合称为类的属性。

域有两种类型——属于每个实例/类的对象或属于类本身。它们分别被称为实例变量和类变量。

类使用 `class` 关键字创建。类的域和方法被列在一个缩进块中。

self

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称，但是在调用这个方法的时候你不为这个参数赋值，Python会提供这个值。这个特别的变量指对象本身，按照惯例它的名称是 `self` 。

虽然你可以给这个参数任何名称，但是 强烈建议 你使用 `self` 这个名称——其他名称都是不赞成你使用的。使用一个标准的名称有很多优点——你的程序读者可以迅速识别它，如果使用 `self` 的话，还有些IDE（集成开发环境）也可以帮助你。

给C++/Java/C#程序员的注释 Python中的 `self` 等价于C++中的 `self` 指针和Java、C#中的 `this` 参考。

你一定很奇怪Python如何给 `self` 赋值以及为何你不需要给它赋值。举一个例子会使此变得清晰。假如你有一个类称为 `MyClass` 和这个类的一个实例 `MyObject` 。当你调用这个对象的方法 `MyObject.method(arg1, arg2)` 的时候，这会由Python自动转为 `MyClass.method(MyObject, arg1, arg2)` ——这就是 `self` 的原理了。

这也意味着如果你有一个不需要参数的方法，你还是得给这个方法定义一个 `self` 参数。

类

一个尽可能简单的类如下面这个例子所示。

```
#!/usr/bin/python
# Filename: simplestclass.py

class Person:
    pass # An empty block

p = Person()
print p
```

（源文件：code/simplestclass.py）

输出

```
$ python simplestclass.py
<__main__.Person instance at 0xf6fcb18c>
```

它如何工作

我们使用 `class` 语句后跟类名，创建了一个新的类。这后面跟着一个缩进的语句块形成类体。在这个例子中，我们使用了一个空白块，它由 `pass` 语句表示。

接下来，我们使用类名后跟一对圆括号来创建一个对象/实例。（我们将在下面的章节中学习更多的如何创建实例的方法）。为了验证，我们简单地打印了这个变量的类型。它告诉我们我们已经在 `__main__` 模块中有了一个 `Person` 类的实例。

可以注意到存储对象的计算机内存地址也打印了出来。这个地址在你的计算机上会是另外一个值，因为Python可以在任何空位存储对象。

对象的方法

我们已经讨论了类/对象可以拥有像函数一样的方法，这些方法与函数的区别只是一个额外的 `self` 变量。现在我们来学习一个例子。

```
#!/usr/bin/python
# Filename: method.py

class Person:
    def sayHi(self):
        print 'Hello, how are you?'

p = Person()
p.sayHi()

# This short example can also be written as Person().sayHi()
```

（源文件：code/method.py）

输出

```
$ python method.py
Hello, how are you?
```

它如何工作

这里我们看到了 `self` 的用法。注意 `sayHi` 方法没有任何参数，但仍然在函数定义时有 `self` 。

__init__方法

在Python的类中有很多方法的名字有特殊的重要意义。现在我们将学习 `__init__` 方法的意义。

`__init__` 方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的 初始化 。注意，这个名称的开始和结尾都是双下划线。

```
#!/usr/bin/python
# Filename: class_init.py

class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print 'Hello, my name is', self.name

p = Person('Swaroop')
p.sayHi()

# This short example can also be written as Person('Swaroop').sayHi()
```

（源文件：code/class_init.py）

输出

```
$ python class_init.py
Hello, my name is Swaroop
```

它如何工作

这里，我们把 `__init__` 方法定义为取一个参数 `name` （以及普通的参数 `self` ）。在这个 `__init__` 里，我们只是创建一个新的域，也称为 `name` 。注意它们是两个不同的变量，尽管它们有相同的名字。点号使我们能够区分它们。

最重要的是，我们没有专门调用 `__init__` 方法，只是在创建一个类的新实例的时候，把参数包括在圆括号内跟在类名后面，从而传递给 `__init__` 方法。这是这种方法的重要之处。

现在，我们能够在我们的方法中使用 `self.name` 域。这在 `sayHi` 方法中得到了验证。

给C++/Java/C#程序员的注释 `__init__` 方法类似于C++、C#和Java中的 constructor 。

类与对象的方法

我们已经讨论了类与对象的功能部分，现在我们来看一下它的数据部分。事实上，它们只是与类和对象的名称空间 绑定 的普通变量，即这些名称只在这些类与对象的前提下有效。

有两种类型的 域 ——类的变量和对象的变量，它们根据是类还是对象 拥有 这个变量而区分。

类的变量 由一个类的所有对象（实例）共享使用。只有一个类变量的拷贝，所以当某个对象对类的变量做了改动的时候，这个改动会反映到所有其他的实例上。

对象的变量 由类的每个对象/实例拥有。因此每个对象有自己对这个域的一份拷贝，即它们不是共享的，在同一个类的不同实例中，虽然对象的变量有相同的名称，但是是互不相关的。通过一个例子会使这个易于理解。

```
#!/usr/bin/python
# Filename: objvar.py

class Person:
    '''Represents a person.'''
    population = 0

    def __init__(self, name):
        '''Initializes the person's data.'''
        self.name = name
        print '(Initializing %s)' % self.name

        # When this person is created, he/she
        # adds to the population
        Person.population += 1

    def __del__(self):
        '''I am dying.'''
        print '%s says bye.' % self.name

        Person.population -= 1

        if Person.population == 0:
            print 'I am the last one.'
        else:
            print 'There are still %d people left.' % Person.population

    def sayHi(self):
        '''Greeting by the person.

        Really, that's all it does.'''
        print 'Hi, my name is %s.' % self.name

    def howMany(self):
        '''Prints the current population.'''
        if Person.population == 1:
            print 'I am the only person here.'
        else:
            print 'We have %d persons here.' % Person.population

swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()

kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

swaroop.sayHi()
swaroop.howMany()
```

（源文件：code/objvar.py）

输出

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
I am the only person here.
(Initializing Abdul Kalam)
Hi, my name is Abdul Kalam.
We have 2 persons here.
Hi, my name is Swaroop.
We have 2 persons here.
Abdul Kalam says bye.
There are still 1 people left.
Swaroop says bye.
I am the last one.
```

它如何工作

这是一个很长的例子，但是它有助于说明类与对象的变量的本质。这里， `population` 属于 `Person` 类，因此是一个类的变量。 `name` 变量属于对象（它使用 `self` 赋值）因此是对象的变量。

观察可以发现 `__init__` 方法用一个名字来初始化 `Person` 实例。在这个方法中，我们让 `population` 增加 `1` ，这是因为我们增加了一个人。同样可以发现， `self.name` 的值根据每个对象指定，这表明了它作为对象的变量的本质。

记住，你只能使用 `self` 变量来参考同一个对象的变量和方法。这被称为 属性参考 。

在这个程序中，我们还看到**docstring**对于类和方法同样有用。我们可以在运行时使用 `Person.__doc__` 和 `Person.sayHi.__doc__` 来分别访问类与方法的文档字符串。

就如同 `__init__` 方法一样，还有一个特殊的方法 `__del__` ，它在对象消逝的时候被调用。对象消逝即对象不再被使用，它所占用的内存将返回给系统作它用。在这个方法里面，我们只是简单地把 `Person.population` 减 `1` 。

当对象不再被使用时， `__del__` 方法运行，但是很难保证这个方法究竟在 什么时候 运行。如果你想要指明它的运行，你就得使用 `del` 语句，就如同我们在以前的例子中使用的那样。

给C++/Java/C#程序员的注释 Python中所有的类成员（包括数据成员）都是 公共的 ，所有的方法都是 有效的 。 只有一个例外：如果你使用的数据成员名称以 双下划线前缀 比如 `__privatevar` ，Python的名称管理体系会有效地把它作为私有变量。 这样就有一个惯例，如果某个变量只想在类或对象中使用，就应该以单下划线前缀。而其他的名称都将作为公共的，可以被其他类/对象使用。记住这只是一个惯例，并不是Python所要求的（与双下划线前缀不同）。 同样，注意 `__del__` 方法与 destructor 的概念类似。

继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过 继承 机制。继承完全可以理解成类之间的 类型和子类型 关系。

假设你想要写一个程序来记录学校之中的教师和学生情况。他们有一些共同属性，比如姓名、年龄和地址。他们也有专有的属性，比如教师的薪水、课程和假期，学生的成绩和学费。

你可以为教师和学生建立两个独立的类来处理它们，但是这样做的话，如果要增加一个新的共有属性，就意味着要在这两个独立的类中都增加这个属性。这很快就会显得不实用。

一个比较好的方法是创建一个共同的类称为 `SchoolMember` 然后让教师和学生的类 继承 这个共同的类。即它们都是这个类型（类）的子类型，然后我们再为这些子类型添加专有的属性。

使用这种方法有很多优点。如果我们增加/改变了 `SchoolMember` 中的任何功能，它会自动地反映到子类型之中。例如，你要为教师和学生都增加一个新的身份证域，那么你只需简单地把它加到 `SchoolMember` 类中。然而，在一个子类型之中做的改动不会影响到别的子类型。另外一个优点是你可以把教师和学生对象都作为 `SchoolMember` 对象来使用，这在某些场合特别有用，比如统计学校成员的人数。一个子类型在任何需要父类型的场合可以被替换成父类型，即对象可以被视作是父类的实例，这种现象被称为多态现象。

另外，我们会发现在 重用 父类的代码的时候，我们无需在不同的类中重复它。而如果我们使用独立的类的话，我们就不得不这么做了。

在上述的场合中， `SchoolMember` 类被称为 基本类 或 超类 。而 `Teacher` 和 `Student` 类被称为 导出类 或 子类 。

现在，我们将学习一个例子程序。

```
#!/usr/bin/python
# Filename: inherit.py

class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: %s)' % self.name

    def tell(self):
        '''Tell my details.'''
        print 'Name:"%s" Age:"%s"' % (self.name, self.age),

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Salary: "%d"' % self.salary

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Marks: "%d"' % self.marks

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 22, 75)

print # prints a blank line

members = [t, s]
for member in members:
    member.tell() # works for both Teachers and Students
```

（源文件：code/inherit.py）

输出

```
$ python inherit.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"22" Marks: "75"
```

它如何工作

为了使用继承，我们把基本类的名称作为一个元组跟在定义类时的类名称之后。然后，我们注意到基本类的 `__init__` 方法专门使用 `self` 变量调用，这样我们就可以初始化对象的基本类部分。这一点十分重要——Python不会自动调用基本类的constructor，你得亲自专门调用它。

我们还观察到我们在方法调用之前加上类名称前缀，然后把 `self` 变量及其他参数传递给它。

注意，在我们使用 `SchoolMember` 类的 `tell` 方法的时候，我们把 `Teacher` 和 `Student` 的实例仅仅作为 `SchoolMember` 的实例。

另外，在这个例子中，我们调用了子类型的 `tell` 方法，而不是 `SchoolMember` 类的 `tell` 方法。可以这样来理解，Python总是首先查找对应类型的方法，在这个例子中就是如此。如果它不能在导出类中找到对应的方法，它才开始到基本类中逐个查找。基本类是在类定义的时候，在元组之中指明的。

一个术语的注释——如果在继承元组中列了一个以上的类，那么它就被称作 多重继承 。

概括

我们已经研究了类和对象的多个内容以及与它们相关的多个术语。通过本章，你已经了解了面向对象的编程的优点和缺陷。Python是一个高度面向对象的语言，理解这些概念会在将来有助于你进一步深入学习Python。

接下来，我们将学习如何处理输入/输出已经如何用Python访问文件。

第12章 输入/输出

目录表

- 文件
- 使用文件
- 储存器
- 储存与取储存
- 概括

在很多时候，你会想要让你的程序与用户（可能是你自己）交互。你会从用户那里得到输入，然后打印一些结果。我们可以分别使用 `raw_input` 和 `print` 语句来完成这些功能。对于输出，你也可以使用多种多样的 `str` （字符串）类。例如，你能够使用 `rjust` 方法来得到一个按一定宽度右对齐的字符串。利用 `help(str)` 获得更多详情。

另一个常用的输入/输出类型是处理文件。创建、读和写文件的能力是许多程序所必需的，我们将会在这章探索如何实现这些功能。

你可以通过创建一个 `file` 类的对象来打开一个文件，分别使用 `file` 类的 `read` 、 `readline` 或 `write` 方法来恰当地读写文件。对文件的读写能力依赖于你在打开文件时指定的模式。最后，当你完成对文件的操作的时候，你调用 `close` 方法来告诉Python我们完成了对文件的使用。

```
#!/usr/bin/python
# Filename: using_file.py

poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
'''

f = file('poem.txt', 'w') # open for 'w'riting
f.write(poem) # write text to file
f.close() # close the file

f = file('poem.txt')
# if no mode is specified, 'r'ead mode is assumed by default
while True:
    line = f.readline()
    if len(line) == 0: # Zero length indicates EOF
        break
    print line,
    # Notice comma to avoid automatic newline added by Python
f.close() # close the file
```

（源文件：code/using_file.py）

输出

```
$ python using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

它如何工作

首先，我们通过指明我们希望打开的文件和模式来创建一个 `file` 类的实例。模式可以为读模式（ `'r'` ）、写模式（ `'w'` ）或追加模式（ `'a'` ）。事实上还有多得多的模式可以使用，你可以使用 `help(file)` 来了解它们的详情。

我们首先用写模式打开文件，然后使用 `file` 类的 `write` 方法来写文件，最后我们用 `close` 关闭这个文件。

接下来，我们再一次打开同一个文件来读文件。如果我们没有指定模式，读模式会作为默认的模式。在一个循环中，我们使用 `readline` 方法读文件的每一行。这个方法返回包括行末换行符的一个完整行。所以，当一个 空的 字符串被返回的时候，即表示文件末已经到达了，于是我们停止循环。

注意，因为从文件读到的内容已经以换行符结尾，所以我们在 `print` 语句上使用逗号来消除自动换行。最后，我们用 `close` 关闭这个文件。

现在，来看一下 `poem.txt` 文件的内容来验证程序确实工作正常了。

储存器

Python提供一个标准的模块，称为 `pickle` 。使用它你可以在一个文件中储存任何Python对象，之后你又可以把它完整无缺地取出来。这被称为 持久地 储存对象。

还有另一个模块称为 `cPickle` ，它的功能和 `pickle` 模块完全相同，只不过它是用C语言编写的，因此要快得多（比 `pickle` 快1000倍）。你可以使用它们中的任一个，而我们在这里将使用 `cPickle` 模块。记住，我们把这两个模块都简称为 `pickle` 模块。

```
#!/usr/bin/python
# Filename: pickling.py

import cPickle as p
#import pickle as p

shoplistfile = 'shoplist.data'
# the name of the file where we will store the object

shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = file(shoplistfile, 'w')
p.dump(shoplist, f) # dump the object to a file
f.close()

del shoplist # remove the shoplist

# Read back from the storage
f = file(shoplistfile)
storedlist = p.load(f)
print storedlist
```

（源文件：code/pickling.py）

输出

```
$ python pickling.py
['apple', 'mango', 'carrot']
```

它如何工作

首先，请注意我们使用了 `import..as` 语法。这是一种便利方法，以便于我们可以使用更短的模块名称。在这个例子中，它还让我们能够通过简单地改变一行就切换到另一个模块（ `cPickle` 或者 `pickle` ）！在程序的其余部分的时候，我们简单地把这个模块称为 `p` 。

为了在文件里储存一个对象，首先以写模式打开一个 `file` 对象，然后调用储存器模块的 `dump` 函数，把对象储存到打开的文件中。这个过程称为 储存 。

接下来，我们使用 `pickle` 模块的 `load` 函数的返回来取回对象。这个过程称为 取储存 。

概括

我们已经讨论了多种类型的输入/输出，及文件处理和使用储存器模块。

接下来，我们将探索异常的概念。

第13章 异常

目录表

当你的程序中出现某些 异常的 状况的时候，异常就发生了。例如，当你想要读某个文件的时候，而那个文件不存在。或者在程序运行的时候，你不小心把它删除了。上述这些情况可以使用异常来处理。

假如你的程序中有一些无效的语句，会怎么样呢？Python会引发并告诉你那里有一个错误，从而处理这样的情况。

考虑一个简单的 `print` 语句。假如我们把 `print` 误拼为 `Print` ，注意大写，这样Python会 引发 一个语法错误。

```
>>> Print 'Hello World'
    File "<stdin>", line 1
      Print 'Hello World'
                        ^
SyntaxError: invalid syntax
>>> print 'Hello World'
Hello World
```

我们可以观察到有一个 `SyntaxError` 被引发，并且检测到的错误位置也被打印了出来。这是这个错误的 错误处理器 所做的工作。

try..except

我们尝试读取用户的一段输入。按**Ctrl-d**，看一下会发生什么。

```
>>> s = raw_input('Enter something --> ')
Enter something --> Traceback (most recent call last):
  File "<stdin>", line 1, in ?
EOFError
```

Python引发了一个称为 `EOFError` 的错误，这个错误基本上意味着它发现一个不期望的 文件尾（由**Ctrl-d**表示）

接下来，我们将学习如何处理这样的错误。

我们可以使用 `try..except` 语句来处理异常。我们把通常的语句放在try-块中，而把我们的错误处理语句放在except-块中。

```
#!/usr/bin/python
# Filename: try_except.py

import sys

try:
    s = raw_input('Enter something --&gt; ')
except EOFError:
    print '\nWhy did you do an EOF on me?'
    sys.exit() # exit the program
except:
    print '\nSome error/exception occurred.'
    # here, we are not exiting the program

print 'Done'
```

（源文件：code/try_except.py）

输出

```
$ python try_except.py
Enter something --&gt;
Why did you do an EOF on me?

$ python try_except.py
Enter something --&gt; Python is exceptional!
Done
```

它如何工作

我们把所有可能引发错误的语句放在 `try` 块中，然后在 `except` 从句/块中处理所有的错误和异常。 `except` 从句可以专门处理单一的错误或异常，或者一组包括在圆括号内的错误/异常。如果没有给出错误或异常的名称，它会处理 所有的 错误和异常。对于每个 `try` 从句，至少都有一个相关联的 `except` 从句。

如果某个错误或异常没有被处理，默认的Python处理器就会被调用。它会终止程序的运行，并且打印一个消息，我们已经看到了这样的处理。

你还可以让 `try..catch` 块关联上一个 `else` 从句。当没有异常发生的时候， `else` 从句将被执行。

我们还可以得到异常对象，从而获取更多有个这个异常的信息。这会在下一个例子中说明。

引发异常

你可以使用 `raise` 语句 引发 异常。你还得指明错误/异常的名称和伴随异常 触发的 异常对象。你可以引发的错误或异常应该分别是一个 `Error` 或 `Exception` 类的直接或间接导出类。

```
#!/usr/bin/python
# Filename: raising.py

class ShortInputException(Exception):
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    s = raw_input('Enter something --&gt; ')
    if len(s) &lt; 3:
        raise ShortInputException(len(s), 3)
    # Other work can continue as usual here
except EOFError:
    print '\nWhy did you do an EOF on me?'
except ShortInputException, x:
    print 'ShortInputException: The input was of length %d, \
          was expecting at least %d' % (x.length, x.atleast)
else:
    print 'No exception was raised.'
```

源文件（code/raising.py）

输出

```
$ python raising.py
Enter something --&gt;
Why did you do an EOF on me?

$ python raising.py
Enter something --&gt; ab
ShortInputException: The input was of length 2, was expecting at least 3

$ python raising.py
Enter something --&gt; abc
No exception was raised.
```

它如何工作

这里，我们创建了我们自己的异常类型，其实我们可以使用任何预定义的异常/错误。这个新的异常类型是 `ShortInputException` 类。它有两个域—— `length` 是给定输入的长度， `atleast` 则是程序期望的最小长度。

在 `except` 从句中，我们提供了错误类和用来表示错误/异常对象的变量。这与函数调用中的形参和实参概念类似。在这个特别的 `except` 从句中，我们使用异常对象的 `length` 和 `atleast` 域来为用户打印一个恰当的消息。

try..finally

假如你在读一个文件的时候，希望在无论异常发生与否的情况下都关闭文件，该怎么做呢？这可以使用 `finally` 块来完成。注意，在一个 `try` 块下，你可以同时使用 `except` 从句和 `finally` 块。如果你要同时使用它们的话，需要把一个嵌入另外一个。

```
#!/usr/bin/python
# Filename: finally.py

import time

try:
    f = file('poem.txt')
    while True: # our usual file-reading idiom
        line = f.readline()
        if len(line) == 0:
            break
        time.sleep(2)
        print line,
finally:
    f.close()
    print 'Cleaning up...closed the file'
```

（源文件：code/finally.py）

输出

```
$ python finally.py
Programming is fun
When the work is done
Cleaning up...closed the file
Traceback (most recent call last):
  File "finally.py", line 12, in ?
    time.sleep(2)
KeyboardInterrupt
```

它如何工作

我们进行通常的读文件工作，但是我有意在每打印一行之前用 `time.sleep` 方法暂停2秒钟。这样做的原因是让程序运行得慢一些（Python由于其本质通常运行得很快）。在程序运行的时候，按**Ctrl-c**中断/取消程序。

我们可以观察到 `KeyboardInterrupt` 异常被触发，程序退出。但是在程序退出之前，finally从句仍然被执行，把文件关闭

概括

我们已经讨论了 `try..except` 和 `try..finally` 语句的用法。我们还学习了如何创建我们自己的异常类型和如何引发异常。

接下来，我们将探索Python标准库。

第14章 Python标准库

目录表

Python标准库是随Python附带安装的，它包含大量极其有用的模块。熟悉Python标准库是十分重要的，因为如果你熟悉这些库中的模块，那么你的大多数问题都可以简单快捷地使用它们来解决。

我们已经研究了一些这个库中的常用模块。你可以在Python附带安装的文档的“库参考”一节中了解Python标准库中所有模块的完整内容。

sys模块

`sys` 模块包含系统对应的功能。我们已经学习了 `sys.argv` 列表，它包含命令行参数。

```
#!/usr/bin/python
# Filename: cat.py

import sys

def readfile(filename):
    '''Print a file to the standard output.'''
    f = file(filename)
    while True:
        line = f.readline()
        if len(line) == 0:
            break
        print line, # notice comma
    f.close()

# Script starts from here
if len(sys.argv) &lt; 2:
    print 'No action specified.'
    sys.exit()

if sys.argv[1].startswith('--'):
    option = sys.argv[1][2:]
    # fetch sys.argv[1] but without the first two characters
    if option == 'version':
        print 'Version 1.2'
    elif option == 'help':
        print '''\
This program prints files to the standard output.
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help'''
    else:
        print 'Unknown option.'
    sys.exit()
else:
    for filename in sys.argv[1:]:
        readfile(filename)
```

（源文件：code/cat.py）

输出

```
$ python cat.py
No action specified.

$ python cat.py --help
This program prints files to the standard output.
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help

$ python cat.py --version
Version 1.2

$ python cat.py --nonsense
Unknown option.

$ python cat.py poem.txt
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

它如何工作

这个程序用来模范Linux/Unix用户熟悉的**cat**命令。你只需要指明某些文本文件的名字，这个程序会把它们打印输出。

在Python程序运行的时候，即不是在交互模式下，在 `sys.argv` 列表中总是至少有一个项目。它就是当前运行的程序名称，作为 `sys.argv[0]` （由于Python从 `0` 开始计数）。其他的命令行参数在这个项目之后。

为了使这个程序对用户更加友好，我们提供了一些用户可以指定的选项来了解更多程序的内容。我们使用第一个参数来检验我们的程序是否被指定了选项。如果使用了 `--version` 选项，程序的版本号将被打印出来。类似地，如果指定了 `--help` 选项，我们提供一些关于程序的解释。我们使用 `sys.exit` 函数退出正在运行的程序。和以往一样，你可以看一下 `help(sys.exit)` 来了解更多详情。

如果没有指定任何选项，而是为程序提供文件名的话，它就简单地打印出每个文件地每一行，按照命令行中的顺序一个文件接着一个文件地打印。

顺便说一下，名称**cat**是 concatenate 的缩写，它基本上表明了程序的功能——它可以在输出打印一个文件或者把两个或两个以上文件连接/级连在一起打印。

`sys.version` 字符串给你提供安装的Python的版本信息。 `sys.version_info` 元组则提供一个更简单的方法来使你的程序具备Python版本要求功能。

```
[swaroop@localhost code]$ python
>>> import sys
>>> sys.version
'2.3.4 (#1, Oct 26 2004, 16:42:40) \n[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)]'
>>> sys.version_info
(2, 3, 4, 'final', 0)
```

对于有经验的程序员， `sys` 模块中其他令人感兴趣的项目有 `sys.stdin` 、 `sys.stdout` 和 `sys.stderr` 它们分别对应你的程序的标准输入、标准输出和标准错误流。

os模块

这个模块包含普遍的操作系统功能。如果你希望你的程序能够与平台无关的话，这个模块是尤为重要的。即它允许一个程序在编写后不需要任何改动，也不会发生任何问题，就可以在Linux和Windows下运行。一个例子就是使用 `os.sep` 可以取代操作系统特定的路径分割符。

下面列出了一些在 `os` 模块中比较有用的部分。它们中的大多数都简单明了。

- `os.name` 字符串指示你正在使用的平台。比如对于Windows，它是 `'nt'` ，而对于Linux/Unix用户，它是 `'posix'` 。
- `os.getcwd()` 函数得到当前工作目录，即当前Python脚本工作的目录路径。
- `os.getenv()` 和 `os.putenv()` 函数分别用来读取和设置环境变量。
- `os.listdir()` 返回指定目录下的所有文件和目录名。
- `os.remove()` 函数用来删除一个文件。
- `os.system()` 函数用来运行shell命令。
- `os.linesep` 字符串给出当前平台使用的行终止符。例如，Windows使用 `'\r\n'` ，Linux使用 `'\n'` 而Mac使用 `'\r'` 。
- `os.path.split()` 函数返回一个路径的目录名和文件名。

  ```
  >>> os.path.split('/home/swaroop/byte/code/poem.txt')
  ('/home/swaroop/byte/code', 'poem.txt')
  ```

- `os.path.isfile()` 和 `os.path.isdir()` 函数分别检验给出的路径是一个文件还是目录。类似地， `os.path.existe()` 函数用来检验给出的路径是否真地存在。

你可以利用Python标准文档去探索更多有关这些函数和变量的详细知识。你也可以使用 `help(sys)` 等等。

概括

我们已经学习了Python标准库中的 `sys` 模块和 `os` 模块的一部分功能。你应该利用Python标准文档去学习这两个模块以及其他模块的更多内容。

接下来，我们将要学习Python中剩余的几个方面的内容，从而使我们的Python课程更加 完整。

第15章 更多Python的内容

目录表

到目前为止，我们已经学习了绝大多数常用的Python知识。在这一章中，我们将要学习另外一些方面的Python知识，从而使我们对Python的了解更加 完整 。

在类中有一些特殊的方法具有特殊的意义，比如 `__init__` 和 `__del__` 方法，它们的重要性我们已经学习过了。

一般说来，特殊的方法都被用来模仿某个行为。例如，如果你想要为你的类使用 `x[key]` 这样的索引操作（就像列表和元组一样），那么你只需要实现 `__getitem__()` 方法就可以了。想一下，Python就是对 `list` 类这样做的！

下面这个表中列出了一些有用的特殊方法。如果你想要知道所有的特殊方法，你可以在《Python参考手册》中找到一个庞大的列表。

名称	说明
init(self,...)	这个方法在新建对象恰好要被返回使用之前被调用。
del(self)	恰好在对象要被删除之前调用。
str(self)	在我们对对象使用 `print` 语句或是使用 `str()` 的时候调用。
lt(self,other)	当使用 小于 运算符（<）的时候调用。类似地，对于所有的运算符（+，>等等）都有特殊的方法。
getitem(self,key)	使用 `x[key]` 索引操作符的时候调用。
len(self)	对序列对象使用内建的 `len()` 函数的时候调用。

单语句块

现在，你已经很深刻地理解了每一个语句块是通过它的缩进层次与其它块区分开来的。然而这在大多数情况下是正确的，但是并非100%的准确。如果你的语句块只包含一句语句，那么你可以在条件语句或循环语句的同一行指明它。下面这个例子清晰地说明了这一点：

```
>>> flag = True
>>> if flag: print 'Yes'
...
Yes
```

就如你所看见的，单个语句被直接使用而不是作为一个独立的块使用。虽然这样做可以使你的程序变得 小一些 ，但是除了检验错误之外我**强**烈建议你不要使用这种缩略方法。不使用它的一个主要的理由是一旦你使用了恰当的缩进，你就可以很方便地添加一个额外的语句。

另外，注意在使用交互模式的Python解释器的时候，它会通过恰当地改变提示符来帮助你输入语句。在上面这个例子中，当你输入了关键字 `if` 之后，Python解释器把提示符改变为 `...` 以表示语句还没有结束。在这种情况下，我们按回车键用来确认语句已经完整了。然后，Python完成整个语句的执行，并且返回原来的提示符并且等待下一句输入。

列表综合

通过列表综合，可以从一个已有的列表导出一个新的列表。例如，你有一个数的列表，而你想要得到一个对应的列表，使其中所有大于2的数都是原来的2倍。对于这种应用，列表综合是最理想的方法。

```
#!/usr/bin/python
# Filename: list_comprehension.py

listone = [2, 3, 4]
listtwo = [2*i for i in listone if i &gt; 2]
print listtwo
```

（源文件：code/list_comprehension.py）

输出

```
$ python list_comprehension.py
[6, 8]
```

它如何工作

这里我们为满足条件（ `if i > 2` ）的数指定了一个操作（ `2*i` ），从而导出一个新的列表。注意原来的列表并没有发生变化。在很多时候，我们都是使用循环来处理列表中的每一个元素，而使用列表综合可以用一种更加精确、简洁、清楚的方法完成相同的工作。

在函数中接收元组和列表

当要使函数接收元组或字典形式的参数的时候，有一种特殊的方法，它分别使用 `*` 和 `**` 前缀。这种方法在函数需要获取可变数量的参数的时候特别有用。

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to specified power.'''
...     total = 0
...     for i in args:
...          total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2, 10)
100
```

由于在 `args` 变量前有 `*` 前缀，所有多余的函数参数都会作为一个元组存储在 `args` 中。如果使用的是 `**` 前缀，多余的参数则会被认为是一个字典的键/值对。

lambda形式

`lambda` 语句被用来创建新的函数对象，并且在运行时返回它们。

```
#!/usr/bin/python
# Filename: lambda.py

def make_repeater(n):
    return lambda s: s*n

twice = make_repeater(2)

print twice('word')
print twice(5)
```

（源文件：code/lambda.py）

输出

```
$ python lambda.py
wordword
10
```

它如何工作

这里，我们使用了 `make_repeater` 函数在运行时创建新的函数对象，并且返回它。 `lambda` 语句用来创建函数对象。本质上， `lambda` 需要一个参数，后面仅跟单个表达式作为函数体，而表达式的值被这个新建的函数返回。注意，即便是 `print` 语句也不能用在lambda形式中，只能使用表达式。

exec和eval语句

`exec` 语句用来执行储存在字符串或文件中的Python语句。例如，我们可以在运行时生成一个包含Python代码的字符串，然后使用 `exec` 语句执行这些语句。下面是一个简单的例子。

```
>>> exec 'print "Hello World"'
Hello World
```

`eval` 语句用来计算存储在字符串中的有效Python表达式。下面是一个简单的例子。

```
>>> eval('2*3')
6
```

assert语句

`assert` 语句用来声明某个条件是真的。例如，如果你非常确信某个你使用的列表中至少有一个元素，而你想要检验这一点，并且在它非真的时候引发一个错误，那么 `assert` 语句是应用在这种情形下的理想语句。当assert语句失败的时候，会引发一个 `AssertionError` 。

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

repr函数

`repr` 函数用来取得对象的规范字符串表示。反引号（也称转换符）可以完成相同的功能。注意，在大多数时候有 `eval(repr(object)) == object` 。

```
>>> i = []
>>> i.append('item')
>>> i
"['item']"
>>> repr(i)
"['item']"
```

基本上， `repr` 函数和反引号用来获取对象的可打印的表示形式。你可以通过定义类的 `__repr__` 方法来控制你的对象在被 `repr` 函数调用的时候返回的内容。

概括

在这一章中，我们又学习了一些Python的特色，然而你可以肯定我们并没有学习完Python的所有特色。不过，到目前为止，我们确实已经学习了绝大多数你在实际中会使用的内容。这些已经足以让你去创建任何程序了。

接下来，我们会讨论一下如何进一步深入探索Python。

第16章 接下来学习什么？

目录表

如果你已经完全读完了这本书并且也实践着编写了很多程序，那么你一定已经能够非常熟练自如地使用Python了。你可能也已经编写了一些Python程序来尝试练习各种Python技能和特性。如果你还没有那样做的话，那么你一定要快点去实践。现在的问题是“接下来学习什么？”。

我会建议你先解决这样一个问题：创建你自己的命令行 地址簿 程序。在这个程序中，你可以添加、修改、删除和搜索你的联系人（朋友、家人和同事等等）以及它们的信息（诸如电子邮件地址和/或电话号码）。这些详细信息应该被保存下来以便以后提取。

思考一下我们到目前为止所学的各种东西的话，你会觉得这个问题其实相当简单。如果你仍然希望知道该从何处入手的话，那么这里也有一个提示。

提示（其实你不应该阅读这个提示） 创建一个类来表示一个人的信息。使用字典储存每个人的对象，把他们的名字作为键。使用cPickle模块永久地把这些对象储存在你的硬盘上。使用字典内建的方法添加、删除和修改人员信息。

一旦你完成了这个程序，你就可以说是一个Python程序员了。现在，请立即寄一封信给我感谢我为你提供了这本优秀的教材吧。是否告知我，如你所愿，但是我确实希望你能够告诉我。

这里有一些继续你的Python之路的方法：

使用Python的**GUI**库——你需要使用这些库来用Python语言创建你自己的图形程序。使用GUI库和它们的Python绑定，你可以创建你自己的IrfanView、Kuickshow软件或者任何别的类似的东西。绑定让你能够使用Python语言编写程序，而使用的库本身是用C、C++或者别的语言编写的。

有许多可供选择的使用Python的GUI：

- **PyQt** 这是Qt工具包的Python绑定。Qt工具包是构建KDE的基石。Qt，特别是配合Qt Designer和出色的Qt文档之后，它极其易用并且功能非常强大。你可以在Linux下免费使用它，但是如果你在Windows下使用它需要付费。使用PyQt，你可以在Linux/Unix上开发免费的（GPL约定的）软件，而开发具产权的软件则需要付费。一个很好的PyQt资源是《使用Python语言的GUI编程：Qt版》请查阅官方主页以获取更多详情。

- **PyGTK** 这是GTK+工具包的Python绑定。GTK+工具包是构建GNOME的基石。GTK+在使用上有很多怪癖的地方，不过一旦你习惯了，你可以非常快速地开发GUI应用程序。Glade图形界面设计器是必不可少的，而文档还有待改善。GTK+在Linux上工作得很好，而它的Windows接口还不完整。你可以使用GTK+开发免费和具有产权的软件。请查阅官方主页以获取更多详情。
- **wxPython** 这是wxWidgets工具包的Python绑定。wxPython有与它相关的学习方法。它的可移植性极佳，可以在Linux、Windows、Mac甚至嵌入式平台上运行。有很多wxPython的IDE，其中包括GUI设计器以及如SPE（Santi's Python Editor）和wxGlade那样的GUI开发器。你可以使用wxPython开发免费和具有产权的软件。请查阅官方主页以获取更多详情。
- **TkInter** 这是现存最老的GUI工具包之一。如果你使用过IDLE，它就是一个TkInter程序。在PythonWare.org上的TkInter文档是十分透彻的。TkInter具备可移植性，可以在Linux/Unix和Windows下工作。重要的是，TkInter是标准Python发行版的一部分。
- 要获取更多选择，请参阅Python.org上的GUI编程wiki页。

不幸的是，并没有单一的标准Python GUI工具。我建议你根据你的情况在上述工具中选择一个。首要考虑的因素是你是否愿意为GUI工具付费。其次考虑的是你是想让你的程序运行在Linux下、Windows下还是两者都要。第三个考虑因素根据你是Linux下的KDE用户还是GNOME用户而定。

未来的章节 我打算为本书编写一或两个关于GUI编程的章节。我可能会选择wxPython作为工具包。如果你想要表达你对这个主题的意见，请加入byte-of-python邮件列表。在这个邮件列表中，读者会与我讨论如何改进本书。

探索更多内容

- **Python**标准库是一个丰富的库，在大多数时候，你可以在这个库中找到你所需的东西。这被称为Python的“功能齐全”理念。我强烈建议你在开始开发大型Python程序之前浏览一下Python标准文档。
- Python.org——Python编程语言的官方主页。你可以在上面找到Python语言和解释器的最新版本。另外还有各种邮件列表活跃地讨论Python的各方面内容。
- **comp.lang.python**是讨论Python语言的世界性新闻组。你可以把你的疑惑和询问贴在这个新闻组上。可以使用Google群在线访问这个新闻组，或加入作为新闻组镜像的邮件列表。
- 《Python实用大全》是一个极有价值的秘诀和技巧集合，它帮助你解决某些使用Python的问题。这是每个Python用户必读的一本书。
- 《迷人的Python》是David Mertz编著的一系列优秀的Python相关文章。
- 《深入理解Python》是给有经验的Python程序员的一本很优秀的书。如果你已经完整地阅读了本书，那么我强烈建议你接下来阅读《深入理解Python》。它覆盖了包括XML处理、单元测试和功能性编程在内的广泛的主题。
- Jython是用Java语言实现的Python解释器。这意味着你可以用Python语言编写程序而同时使用Java库！Jython是一个稳定成熟的软件。如果你也是一个Java程序员，我强烈建议你尝试一下Jython。
- IronPython是用C#语言实现的Python解释器，可以运行在.NET、Mono和DotGNU平台上。这意味着你可以用Python语言编写程序而使用.NET库以及其他由这三种平台提供的库！IronPython还只是一个前期alpha测试软件，现在还只适合用来进行试验。Jim Hugunin，IronPython的开发者，已经加入了微软公司，将在将来全力开发一个完整版本的IronPython。
- Lython是Python语言的Lisp前段。它类似于普通的Lisp语言，会被直接编译为Python字节码，这意味着它能与我们普通的Python代码协同工作。
- 另外还有很多很多的Python资源。其中比较有趣的有Daily Python-URL!，它使你保持与Python的最新进展同步。另外还有Vaults of Parnassus、ONLamp.com Python DevCenter、dirtSimple.org、Python Notes等等。

概括

现在，我们已经来到了本书的末尾，但是就如那句名言，这只是 开始的结束 ！你现在是一个满怀渴望的Python用户，毫无疑问你准备用Python解决许多问题。你可以使你的计算机自动地完成许多先前无法想象的工作或者编写你自己的游戏，以及更多别的什么东西。所以，请出发吧！

附录A 自由/开放源码软件（FLOSS）

FLOSS基于社区的概念，而它本身基于共享，特别是知识共享的概念。FLOSS可以免费使用、修改和再发行。

如果你已经读了本书，那么你一定熟悉FLOSS，因为你一直在使用**Python**！

如果你想要了解更多的FLOSS，你可以探索下面这个列表中的软件。我列出了一些最著名的FLOSS以及那些可以跨平台（即在Linux、Windows等）工作的FLOSS。这样你无需马上切换到Linux就可以尝试使用这些软件了， 尽管你最终一定会转到Linux上的 。

- **Linux** 这是一个正在慢慢被世界接纳的FLOSS操作系统！它最初由Linus Torvalds在学生时候开发。现在，它已经可以与微软Windows相匹敌。最新的2.6版本核心，无论从速度、稳定性还是扩展性角度来说，都是一个巨大的突破。【Linux核心】

- **Knoppix** 这是一个仅仅在CD上运行的Linux发行版！它不需要安装——你只需要重新启动你的计算机，把CD放入光驱，就可以开始使用一个完全的Linux发行版了！你可以使用所有的随标准Linux发行版发行的FLOSS，如运行Python程序、编译C程序、看电影等等。然后再次重启你的计算机，取出CD，就可以使用你现有的操作系统了，就好像什么都没有发生过一样。【Knoppix】

- **Fedora** 这是一个由社区开发维护的发行版，由Red Hat公司赞助。它是最流行的Linux发行版之一。它包含Linux核心、KDE、GNOME和XFCE桌面以及众多的FLOSS，而所有这些都易于安装、易于使用。

  如果你担心你是一个完全的Linux生手，那么我推荐你尝试**Mandrake Linux**。最新发布Mandrake 10.1确实很棒。【Fedora Linux、Mandrake Linux】

- **OpenOffice.org** 这是一个优秀的办公套件，它基于Sun Microsystems的StarOffice软件。OpenOffice由文本编写器、演讲辅助、电子表格和绘图组件等等组成。它甚至可以方便地打开和编辑微软Word和PowerPoint文件。它可以在几乎所有平台上运行。即将推出的OpenOffice 2.0有一些重大的改进。【OpenOffice】

- **Mozilla Firefox** 这是被认为可以在未来几年击败Internet Explorer（仅按照市场份额计算）的下一代网络浏览器。它极快，它的一些合理的、令人印象深刻的特性广受好评。它的扩展理念允许在它上面添加各种功能。

  它的姐妹产品**Thunderbird**是一个优秀的电子邮件客户端，使阅读电子邮件变得十分快捷。【Mozilla Firefox、Mozilla Thunderbird】

- **Mono** 这是一个微软.NET平台的开源实现。它使我们可以在Linux、Windows、FreeBSD、Mac OS和许多其他平台上创建和运行.NET程序。Mono执行CLI和C#的ECMA标准，这个标准已经由微软、英特尔和惠普提交称为一个开放标准。这也是迈向

ISO标准的一步。

目前，Mono包含一个完整的C#主控制台（它本身也由C#编写！）、一个具备完整特性的ASP.NET实现、许多数据库ADO.NET提供器另外还有每天不断改善和增加的新特性。【Mono、ECMA、Microsoft .NET】

- **Apache**网络服务器 这是最流行的开源网络服务器。事实上，它是地球上最流行的网络服务器！它运行着几乎60%的网站。对——Apache处理的网站比它所有的竞争对手（包括微软IIS）之和还要多。【Apache】
- **MySQL** 这是一个极其流行的开源数据库服务器。它以它的快速最为著名。在它的最新版本中又添加了更多的特性。【MySQL】
- **MPlayer** 这是一个视频播放器，可以播放DivX、MP3、Ogg、VCD、DVD……谁说开源软件就不能具有趣味呢？【MPlayer】
- **Movix** 这是一个Linux发行版，它基于Knoppix仅仅在CD上运行用来播放电影！你可以创建Movix的CD。它们是可启动的CD，当你重启计算机的时候，放入CD，电影就会自己开始播放！使用Movix观看电影，你甚至不需要硬盘。【Movix】

上面这个列表只是希望给你一个大概的印象——还有很多别的优秀FLOSS，比如Perl语言、PHP语言、Drupal网站内容管理系统、PostgreSQL数据库服务器、TORCS赛车游戏、KDevelop IDE、Anjuta IDE、Xine——电影播放器、VIM编辑器、Quanta+编辑器、XMMS音频播放器、GIMP图像编辑程序……这个列表可以一直继续下去。

访问下述网站以获取更多FLOSS信息：

- SourceForge
- FreshMeat
- KDE
- GNOME

要获知FLOSS世界的最新进展，请访问下述网站：

- OSNews
- LinuxToday
- NewsForge
- SwaroopCH's blog

那么，现在就出发去探索广博、免费、开放的FLOSS世界了吧！

附录B 关于本书

目录表

我在编写本书时使用的几乎所有软件都是 免费开放源码的软件 。在编写本书的第一个草稿的时候，我使用的是Red Hat 9.0 Linux，而现在第六次改写的时候，使用的是Fedora Core 3 Linux。

最初，我使用KWord编写本书（在前言的本书的由来中已经介绍了）。后来，我开始使用DocBook XML和Kate，但是我发现这样太乏味。所以，我开始使用OpenOffice，它对格式的控制以及生成PDF的能力是很棒的。但是它生成的HTML过于庞大。最后，我发现了XEmacs，于是我又开始重新使用DocBook XML来编写本书，并且那时我打算把这个模式作为将来长期的方案。在这个最新的第六次重写时，我决定使用Quanta+来编辑。

我使用了标准的XSL样式表，它随Fedora Core 3 Linux附带。另外，我也使用了标准的默认字体。我编写了一个CSS文件来为HTML页增加颜色和样式。同时，我还用Python语言编写了一个粗劣的词汇分析器，它自动为书中所有的程序进行语法加亮。

关于作者

Swaroop C. H. 在Yahoo!驻印度班加罗尔的办事处工作，他十分热爱他的工作。他目前在技术领域的兴趣有：包括Linux、DotGNU、Qt和MySQL在内的FLOSS、Python和C#编程语言。另外他在业余时间编写一些如本书这样的教材和其他软件，以及编写他的网上日记。他的其他爱好有咖啡、Robert Ludlum的小说、远足和政治等。

如果你有兴趣了解他的更多故事，可以在www.swaroopch.info上查看他的网上日记。

关于译者

沈洁元 目前是上海交通大学无线通信研究所的一名硕士研究生。他现在的研究领域主要在多载波CDMA系统的同步、信道估计、多用户检测等方面。Python语言（和Numeric库）是他目前在进行仿真和其他科研工作时使用的主要编程语言。在业余时间，他乐衷于各种FLOSS，如FreeBSD操作系统、PyGTK等等。电影、F1赛车和网球也是他的兴趣爱好。

关于简体中文译本

我在半年多前开始学习使用Python编程语言。正如Swaroop在本书中所说的那样，它很快就成为“我最喜欢的编程语言”。目前我的几乎所有编程工作都使用Python。从我的切身体会来说，Python最大的特点就是易懂、易用、高效率。我相信，如果你已经学完了本书，并且尝试着编写了一些程序后，你一定会有相同的感受。

Swaroop C. H.的这本书是我学习Python时的第一本教材。它简单明晰，可以在最短的时间内把你领进Python的世界。它不是很长，但是覆盖了几乎所有重要的Python知识。在第一次读本书的时候，我就深切的感到这是给Python初学者的一本极佳教材，应该是每一位Python初学者的第一本教材。

我利用业余时间翻译了这本教材的简体中文译本。一方面是为了感谢Swaroop给我们带来了那么好的一本教材，同时也是为了把本书介绍给更多的中国读者，希望让Python在中国更加普及。如果读了本书之后，你开始将Python应用于你的工作学习，这将是我和Swaroop以及其他Python用户的荣幸。如果你在学习和使用Python的过程中，遇到任何问题，你一定要试试使用Python的邮件列表资源。你一定会得到世界各地的Python高手的热情帮助。

本书的英文原名为《A Byte of Python》。经过与Swaroop的探讨，在翻译时，我把书名定为《简明 Python 教程》，以充分体现本书区别于其他Python教材的鲜明特色。在翻译这本简体中文译本时，我力求准确清晰。在原书中个别不甚清晰的地方，都与作者进行讨论后再行翻译。另外，在这本简体中文译本中，我还为书中所有的程序例子配上了源代码，并且在书后附上了中英对照的术语表，以便读者以后继续学习其他Python英文资料。

本译本作为原书的派生作品，依照创作公用约定（署名-非派生作品-非商业用途）发布。简单地说，你只要署上我的名字，就可以免费复制、分发和展示本译本。未得到我的允许，你禁止把本译本用于商业目的，也不能再在本译本的基础上修改、派生新的作品。

如果你对本书和译本有任何批评和建议，十分欢迎你与我联系：orion_val@163.com。

附录C 修订记录

目录表

本文档在2005年1月13日4点02分生成。

修订记录

1.20版	2005年1月13日
使用FC3上的Quanta+的完全重写。做了许多修正和更新。添加了许多新的例子。重写了我的DocBook设置。	
1.15版	2004年3月28日
少量修订。	
1.12版	2004年3月16日
添加修正了一些内容。	
1.10版	2004年3月9日
感谢我的热情读者的帮助，我对更多的笔误做了修改。	
1.00版	2004年3月8日
在从读者处获得了大量反馈和建议之后，我对本书的内容做了重要的修订，并且改正了一些笔误。	
0.99版	2004年2月22日
增加了模块一章。增加了对可变数目函数参数的详细介绍。	
0.98版	2004年2月16日

编写了一个Python脚本和CSS样式表来改善XHTML的输出效果。其中包括一个功能还很拙劣的词汇分析器，用来自动地为程序做类似于VIM地语法加亮。	
0.97版	2004年2月13日
（再次）使用DocBook XML完全重写。本书改进了许多——更加有条理和易读。	
0.93版	2004年1月25日
增加了关于IDLE的介绍以及更多Windows®相关的话题。	
0.92版	2004年1月5日
修改了几个例子。	
0.91版	2003年12月30日
修正了排版错误。改进了许多章节的内容。	
0.90版	2003年12月18日
增加了2章。使用OpenOffice格式修订。	
0.60版	2003年11月21日
完全地重写和扩展。	
0.20版	2003年11月20日
修改了一些排版错误和其他错误。	
0.15版	2003年11月20日
改用DocBook XML。	
0.10版	2003年11月14日
最初使用KWord编写的草稿。	

术语表

argument	实参
attribute	属性
base class	基本类
block	块
character	字符
class	类
comment	注释
complex number	复数
derived class	导出类
dictionary	字典
escape sequence	转义符
exception	异常
expression	表达式
field	域
float	浮点数
function	函数
identifier	标识符
indentation	缩进
indexing	索引
instance	实例
integer	整数
list	列表
list comprehension	列表综合
literal constant	字面意义上的常量
logical line	逻辑行
long integer	长整数
method	方法
module	模块

namespace	名称空间
object	对象
operand	操作数
operator	运算符
parameter	形参
pickle	储存器
physical line	物理行
sequence	序列
shebang line	组织行
slicing	切片
statement	语句
string	字符串
subclass	子类
superclass	超类
tuple	元组
type	类型
variable	变量